UNIVERSITÄT OSNABRÜCK

Rechenzentrum

Einführung in JAVA

Einführung in JAVA

Herausgeber:
Der Präsident der Universität Osnabrück

Universität Osnabrück
- Rechenzentrum -
Albrechtstraße 28
D-49076 Osnabrück

Wolfgang.Meyer@rz.Uni-Osnabrueck.DE

Version: 0.1, 2/98

Druck:
Hausdruckerei der Universität

Inhaltsverzeichnis

1 Einleitung

Das vorliegende Skript soll einen ersten Einblick in die neue objekt-orientierte Programmiersprache Java geben. Kenntnisse einer Programmiersprache, vorzugsweise C, werden vorausgesetzt. Kenntnisse in objekt-orientierter Programmierung sind hilfreich, aber nicht unbedingt erforderlich.

Diese Skript kann und will keine vollständige Sprachbeschreibung sein, sondern möchte in möglichst einfacher, verständlicher und kompakter Form wichtige Elemente von Java vorstellen. Deshalb wurde an einigen Stellen auf eine absolut präzise Formulierung zugunsten der Verständlichkeit verzichtet.

Auf eine Referenz der umfangreichen Klassenbibliothek von Java wurde ebenfalls verzichtet. Allein die Beschreibung der Klassen würde den Umfang dieses Skript sprengen. Bei Bedarf (und den hat man) konsultiere man die Literatur [2].

☺ Vorsicht: Java macht süchtig!

1.1 Historie

Mit der Entwicklung von Java wurde etwa 1990 bei SUN Microsystems im Rahmen des Green-Projekts begonnen. Eine Gruppe um James Goshling und Bill Joy verfolgte zunächst das Ziel, Fernsehen interaktiv zu gestalten. (*Consumer Electronics*, Set-Top Boxen)

Ziel war es, von Anfang an eine Sprache zu verwenden, die plattform-unabhängig, objekt-orientiert, einfach ist und natürlich die Kommunikation im Netzwerk unterstützt. Etwa 1992 wurde die Sprache *Oak* (später in Java umbenannt) vorgestellt, in die Elemente von C++, C, objective C, Eiffel und Smalltalk eingingen.

Mitte 1994 ging das Interesse der Gruppe mehr und mehr auf das Web über. Im Herbst 1994 stellte die Gruppe den Web-Browser *Web-Runner* vor, der mit Hilfe der Java Technologie entwickelt wurde.

1995 stellte SUN auf der *SUN World* in San Francisco Java und den Browser *HotJava* vor. 1997 wurde die Version 1.1.1 des Java Development Toolkit (JDK) vorgestellt, das alle nötigen Werkzeuge für die Java Programmierung enthält. Im Moment (2/98) ist JDK 1.1.5 aktuell; die Version 1.2 befindet sich im Beta Stadium. Auf der *International Internet Associate Symposium* im November 1997 in Berlin wurde von SUN angekündigt, daß das JDK eingefroren werden soll.

Mittlerweile haben viele Firmen (IBM, NetScape, Borland, Oracle, Micro$oft, Adobe, Lotus, u.a.) Java lizensiert, um eigene Anwendungen damit entwickeln zu können.

1.2 Eigenschaften von Java

Die neue Programmiersprache Java zeichnet sich u.a durch folgende Eigenschaften aus. Java ist:

- einfach
  In Java sind ausgesuchte Konzepte eingegangen (Objekte, Schnittstellen, Ausnahmen, Threads) und es wurde versucht, den Sprachumfang möglichst überschaubar zu halten. Die Sprache verzichtet auf Zeigerarithmetik, Präprozessoranweisungen, Überladung von Operatoren, automatische Typkonvertierung, wie sie von C++ bzw. C her bekannt sind.

- objekt-orientiert
  Im Gegensatz zu C++, die eher als Hybridsprache einzustufen ist, ist Java vollständig objekt-orientiert. Java besitzt umfangreiche Klassenbibliotheken (*Application Programming Interfaces, APIs*) u.a. auch zur Programmierung graphischer Benutzeroberflächen (*Graphical User Interfaces, GUIs*) unter Verwendung der Klassen des *Abstract Window Toolkit* (AWT).

- robust
  Zeiger und Zeigerarithmetik können in Java nicht verwendet werden. Damit entfallen häufige Fehlerquellen, wie man sie von C++ oder C her kennt. Vom Programm nicht mehr erreichbare Objekte werden automatisch vom System durch *Garbage Collection* beseitigt.
  Zur Fehlerbehandlung von Java gehören Ausnahmen (*Exceptions*). Der Programmierer sollte durch das Abfangen dieser Ausnahmen dafür sorgen, daß sein Programm auch im Fehlerfall korrekt weiter arbeiten kann.

- compiliert
  Java Programme werden vom Java Compiler (*javac*) in Byte-Code übersetzt, der dann von der Virtuellen Java Maschine *(java*) interpretativ verarbeitet werden kann.

- plattform-unabhängig
  Nur die Virtuelle Java Maschine muß für jede Plattform neu hergestellt werden (Bei Java fähigen WWW-Browsern ist diese mit integriert). Die Byte-Code Dateien bleiben davon unberührt. Daher ist es möglich, Programme in übersetzter Form zu verteilen.

- dynamisch
  Java verwendet dynamisches Binden für Klassen und Methoden. Daher können einzelne Klassen unabhängig voneinander neu übersetzt werden.

- erweiterbar
  Java erlaubt die Implementierung von Methoden in anderen Sprachen.

- multi-threaded
  Mit Hilfe von Threads können in Java Abläufe parallelisiert werden. Es gibt Synchronisations- und Schedulingmechanismen.

- WWW-geeignet
  Durch Java's multimediale Fähigkeiten und durch Applet Programmierung eignet sich Java hervorragend für das Web. (Komponentenmodell, *Java Beans*)

1.3 Exkurs: Objekt-orientierte Programmierung

Objekt-orientierte Programmiersprachen verwenden im Gegensatz zu den herkömmlichen prozeduralen Programmiersprachen moderne Konzepte. U.a. erzielen objekt-orientierte Programmiersprachen eine hohe Wiederverwendbarkeit von Code durch die Definition von Klassen.

Eine *Objekt* ist eine *gekapselte Datenstruktur*, die neben ihren *Eigenschaften* (Variablen) auch ihre *Methoden* (Funktionen), die auf die Eigenschaften wirken, enthält.

Auch die reale Welt kann als eine Ansammlung von Objekten betrachtet werden. Nehmen wir als Beispiel ein konkretes Auto, einen `VWPolo`. Dieses Auto hat eine Menge an Eigenschaften, z.B. die Farbe, PS-Zahl, momentane Geschwindigkeit etc., also

```
VWPolo.Farbe = "blau";
VWPolo.PS = 50;
VWPolo.Geschwindigkeit = 80.5;
```

Daneben gibt es eine Menge von Methoden, die auf die momentanen Eigenschaften wirken können:

```
VWPolo.GasGeben();
VWPolo.Bremsen();
```

Die obigen beiden Methoden wirken auf die Eigenschaft Geschwindigkeit des Objekts `VWPolo`.

Objekte sind *Instanzen* (Ausprägungen) von *Klassen.*

In unserem Beispiel wäre das Objekt `VWPolo` eine Instanz der Klasse `Auto`. Klassen können Unterklassen haben, wobei die Methoden und Eigenschaften der Oberklassen an die Unterklasse weitergegeben (vererbt) werden.

In Pseudocode könnte das folgendermaßen (vereinfacht) aussehen:

```
class Auto {
        String Farbe;
        int PS;
        float Geschwindigkeit;

        function GasGeben() {
                // wie wirkt sich GagGeben aus
        }

        function Bremsen() {
                // wie wirkt sich Bremsen aus
        }
}
```

Damit könnte die Klasse `Auto` definiert sein. Das Objekt `VWPolo` könnte man dann erzeugen über:

```
        Auto VWPolo = new Auto();

        VWPolo.Geschwindigkeit = 40.0;
```

Java kennt keine Mehrfachvererbung, wie es z.B. von C++ her bekannt ist. D.h eine bestimmte Klasse in Java kann nur vor einer Klasse abstammen.

1.4 Grundlegende Unterschiede von JDK 1.0 zu JDK 1.1

Im folgenden werden kurz grundlegende Neuerungen des JDK 1.1 skizziert. Abwärts-Kompatibilität ist zwar gewährleistet, wird aber vom Compiler angemerkt. Außerdem hält sich SUN bedeckt, wie lange die Kompatibilität erhalten bleibt:

- grundsätzlich anderes Event-Handling
  Graphische Benutzeroberflächen arbeiten Ereignis orientiert. D.h. auf das Eintreffen bestimmter Ereignisse (*Events*) wird mit der Ausführung bestimmter Aktionen (Methoden) geantwortet. Solche Ereignisse können z.B. Mausklick, Fenster vergrößern etc. sein.
  Im JDK 1.0 gab es ein Objekt vom Typ `Event`, aus dem alle relevanten Informationen extrahiert werden mußten. Im JDK 1.1 handelt es sich um einen delegations-basierten Event-Transport: Es gibt mehrere Event Klassen, die an ein Zuhörer-*Objekt (Event Listener*) weitergeleitet werden. Der Event Listener entscheidet über die weitere Behandlung des Events.
  Das Event-Handling des JDK 1.1 wird z.B. von Netscape Communicator, Version 4.04 unterstützt.

- Clipboard - Unterstützung wird möglich. (Cut and Paste)
- Pop-Up Menüs und Short Cuts.
- Druckausgabe wird möglich.
  (siehe auch Beispiele im Anhang).
- Namensänderung bei einigen Methoden.

Doch bevor wir uns weiter mit Java beschäftigen, wollen wir mit einigen Beispielen beginnen.

2 Elementare Java Programme

Es hat sich in der EDV eingebürgert, Programmiersprachen mit einem Programm vorzustellen, das den String „Hello, World“ ausgibt. An diese Tradition wollen wir anknüpfen.

Doch zunächst eine Vorbemerkung. Java Programme können grob in zwei Kategorieen eingeteilt werden:

- Applikationen sind eigenständige (*stand-alone*) Java-Programme, die ohne Hilfe eines WWW Browsers auskommen. Applikationen können eine graphische Oberfläche haben (siehe Beispiel 1 und 2).
- Applets sind (uneigenständige) Java-Programme, die mit Hilfe des `<APPLET>` tags in HTML Dokumenten von einem WWW Browser geladen und ausgeführt werden. (Beispiel 3)

Beispiel 1:

```
public class HelloWorld1 {
  public static void main (String argv []) {
    System.out.println ("Hello, World");
  }
}
```

Im Beispiel 1 wird eine Klasse mit Namen `HelloWorld1` definiert, die `public` ist und die nur eine Methode, nämlich `main()` besitzt. `Public` Klassen sind auch von Klassen erreichbar, die sich in anderen Quelldateien befinden. Weitere Sichtbarkeitskriterien sind: default (d.h. ohne jegliches Schlüsselwort), `private` und `protected`, die später erklärt werden.

Die einzige Methode `main()` ist ebenfalls `public` und hat als Typ `void`, d.h. es wird kein Wert zurückgeliefert. Wird eine Methode `static` definiert, so bedeutet dieses, daß für alle Instanzen dieser Klasse die Methode nur einmal existiert.

Als Parameter hat `main()` ein Array von Strings. Der Rumpf der Methode besteht aus nur einer Anweisung, die den String „Hello, World“ auf die Standardausgabe (Bildschirm) ausgibt. Genauer gesagt wird hier die Methode `println()` des `out` Objekts aufgerufen, das zum Objekt `System` gehört.

Die Klassen `System` und `String` gehören zum `java.lang` Paket, das automatisch verfügbar ist.

Nachdem man Beispiel 1 in eine Datei mit dem Namen `HelloWorld1.java` eingetragen hat, kann eine Übersetzung durch den Befehl

```
javac HelloWorld1.java
```

geschehen. Das Kompilat wird in eine Datei namens `HelloWorld1.class` geschrieben. Gestartet wird die Java Applikation mit

```
java HelloWorld1
```

Danach sollte der String „Hello, World“ auf der Standardausgabe erscheinen.

Beispiel 2:

```
import java.awt.*;

/**
  * Dies ist die Klasse HelloWorld2
  * zur Demonstration einer Applikation
  * mit einer graphischen Oberfläche */

public class HelloWorld2 extends Frame {

    /** Hier wird die main Methode implementiert */
    public static void main (String argv []) {

    Frame myFrame = new HelloWorld2();
    List  myList  = new List(5,false);

    myFrame.setTitle(myFrame.getClass().getName());
    myList.addItem("Hello, World");
    System.out.println("Hello, World");
    myFrame.add(myList);
    myFrame.pack();
    myFrame.show();
  }

  /** Hier wird ein Event Handler ueberschrieben */
  public boolean handleEvent(Event evt) {
    if (evt.id == Event.WINDOW_DESTROY)
      System.exit(0);
    System.out.println(evt.id);
    return true;
  }
}
```

Im 2. Beispiel handelt es sich um eine Applikation mit einer graphischen Oberfläche. Oberflächen werden mit dem Abstract Window Toolkit (AWT) programmiert, das zunächst mit einer `import` Anweisung verfügbar gemacht werden muß.

Im folgenden wird eine Klasse `HelloWorld2` definiert und als Unterklasse von `Frame`, der Klasse der Fenster, festgelegt. `HelloWorld2` umfaßt zwei Methoden. Die `main()` Methode wird beim Start des Programms automatisch aufgerufen und die Methode `handleEvent()`. Die Methode `handleEvent()` ist bereits in der Klasse `Component` definiert und wird in der vorliegenden Klasse überschrieben.

In der Methode `main()` wird zunächst ein Fenster `myFrame` und eine Komponente (d.h. grob gesprochen ein Element des Fensters) `myList` erzeugt. Es handelt sich hierbei um neue Objekte, die mit dem Schlüsselwort `new` generiert werden. Objekte verschwinden automatisch (durch Garbage Collection) falls sie vom System nicht mehr erreichbar sind.

Danach werden eine Reihe von Methoden aufgerufen , die sich auf die Objekte `myFrame` bzw. `myList` beziehen und von der Klasse Frame oder von Oberklassen von `Frame` ererbt wurden.

`myFrame,setTitle()` setzt einen Titel des Fensters, `myList.addItem()` fügt einen String der Liste hinzu, `myFrame.add()` fügt die angegebene Komponente dem Fenster hinzu, `myFrame.pack()` ordnet die Komponenten im Fenster an und `myFrame.show()` zeigt schließlich das fertige Fenster an.

Über die Anweisung `myList.addItem("Hello, World");` wird der String `"Hello, World"` ausgegeben. Genauer gesagt in eine Liste, die dem Fenster hinzugefügt wurde. Und über `System.out.println("Hello, World");` wird der String auf die Standardausgabe ausgegeben.

Ereignis-gesteuerte Programmierung ist der Hauptbestandteil bei der Programmierung graphischer Oberflächen. Der Benutzer löst durch bestimmte Aktionen (z.B. Maustaste drücken, Focus auf ein Fenster bringen, Fenster vergrößern, Taste drücken) bestimmte Ereignisse aus. Das Programm hat dann die Möglichkeit auf das Eintreffen bestimmter Ereignisse gewisse Aktionen auszulösen.

In diesem Beispiel wird das Event-Handling von JDK 1.0 beschrieben. Durch die Methode `handleEvent()` der Klasse `HelloWorld2` wird das Ereignis `Event.WINDOW_DESTROY` behandelt. Trifft dies Ereignis ein, wird über `System.exit()` die gesamte Applikation beendet. Andere `Event.id` werden auf die Standardausgabe ausgegeben. (Zum Event Handling siehe auch Kapitel 6.)

Das Beispiel 2.1 zeigt das Event-Handling im JDK 1.1. Die Klasse `HelloWorld21` implementiert einen `WindowListener`. Im Konstruktor der Klasse wird der `WindowListener` an die Klasse `HelloWorld21` gebunden.

Bei der Erzeugung von Objekten werden implizit Konstruktoren der Klassen aufgerufen. Konstruktoren heißen so wie die Klasse und liefern keinen Wert zurück. Häufig werden Intialisierungen durch Konstruktoren vorgenommen. Fehlt bei der Definition einer Klasse der entsprechende Konstruktor, so wird der Konstruktor der Oberklasse aufgerufen.

Im weiteren Verlauf des Beispiels werden die Methoden des `WindowsListener` implementiert. Für uns ist dabei im Moment nur die Methode `windowClosing()` von Interesse, die die Applikation beendet.

Beipiel 2.1

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

/**
  * Dies ist die Klasse HelloWorld21
  * zur Demonstration einer Applikation
  * mit einer graphischen Oberfläche */

public class HelloWorld21 extends Frame implements WindowListener {
```

```
    public HelloWorld21(){
      addWindowListener(this);
    }

    /** Hier wird die main Methode implementiert */
    public static void main (String argv []) {

    Frame myFrame = new HelloWorld21();
    List  myList  = new List(5,false);

    myFrame.setTitle(myFrame.getClass().getName());
    myList.addItem("Hello, World");
    System.out.println("Hello, World");
    myFrame.add(myList);
    myFrame.pack();
    myFrame.show();
  }

  public void windowClosing(WindowEvent e) {
    System.out.println("Ende ...");
    System.exit(0);
  }

   public void windowClosed(WindowEvent event) {
         }
   public void windowDeiconified(WindowEvent event) {
         }
   public void windowIconified(WindowEvent event) {
         }
   public void windowActivated(WindowEvent event) {
         }
   public void windowDeactivated(WindowEvent event) {
         }
   public void windowOpened(WindowEvent event) {
         }

}
```

Übersetzt und gestartet werden können Beispiele 2 und 2.1 analog zum Beispiel 1. Nach dem Start erscheint ein neues Fenster mit dem String `"Hello, World"`. Würden in den Beispielen 2 und 2.1 das Event-Handling nicht implementiert sein, so könnte die Applikation nur mit Gewalt (`^C`) beendet werden.

Das obige Beispiel erzeugt das folgende Fenster:

Das Beispiel 3 demonstriert ein Applet, das von Java fähigen WWW Browsern geladen und ausgeführt werden kann. In diesem Fall entfällt der Aufbau einers Hauptfensters, da dieses schon durch den Browser vorhanden ist. Applets sind Unterklassen von `Applet`.

Beispiel 3

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

public class HelloWorld3 extends Applet {
```

```
  public void init() {
    System.out.println ("Hello, World");
  }

  public void paint (Graphics g) {
    g.drawString("Hello, World", 100, 50);
  }
}
```

Die Klasse `HelloWorld3` besteht aus zwei Methoden. Die Methode `init()` gibt den String `"Hello, World"` auf die Standardausgabe aus. Das ist in diesem Fall die sogenannte Java Konsole, die in Netscape über `Options -> Show Java Console` aufgeblendet werden kann. Die Methode `init()` wird automatisch aufgerufen, wenn das Applet initialisiert wird. Es gibt weitere Meilensteine im Leben eines Applets, die später besprochen werden.

Mit `paint()` wird ein graphischer Kontext (z.B. ein Text) dargestellt. `paint()` wird z.B. dann automatisch aufgerufen, wenn das Applet erstmailg dargestellt wird. `paint()` gibt den String `"Hello,World"` an die Stelle (100,100) (in Pixel) aus.

Nachdem Beispiel 3 in eine Datei namens `HelloWorld3.java` eingetragen wurde, kann mit dem Befehl

```
javac HelloWorld3.java
```

diese Datei übersetzt werden. Dargestellt werden kann dieses Applet entweder mit einem Java fähigen WWW Browser (HotJava, Netscape, Internet Explorer ...) oder mit dem `appletviewer` (der Appletviewer ist Teil des JDK), indem man die folgende HTML Datei läd:

```
<html>
<head>
</head>
<body>
Applet: HelloWorld3
<p>
<applet code="HelloWorld3.class" width=400 height=300>
</applet>
</body>
</html>
```

Mit Hilfe des `<applet>` tags wird die übersetzte Datei `HelloWorld3.class` geladen und vom Browser ausgeführt. Alternativ kann mit dem Appletviewer über

```
appletviewer <Name der HTML Datei>
```

das Applet gestartet werden.

Nun stellt sich die Frage: Welche Aufgaben können mit Java gelöst werden? Was macht Java so interessant?

Nun, zunächst einmal kann man Probleme, die mit herkömmlichen Programmiersprachen lösbar sind, auch mit Java behandeln. Darüber hinaus zeichnet sich Java u.a. aus durch:

- multi-mediale Fähigkeiten
- kompilierte Programme über das Netz zu laden und lokal auszuführen und damit eng verbunden seine
- Web Fähigkeiten (CGI, Servlets, Java-JavaScript Schnittstelle)
- Datenbankanbindung über JDBC.
- Komponentenmodell (Java Beans).

Natürlich dürfen auch Sicherheitsaspekte nicht zu kurz kommen. Ein Applet kann z.B. nicht auf das Dateisystem des lokalen Rechners zugreifen und nur eine Netzverbindung zu dem Rechner aufnehmen, vom dem es geladen wurde. Außerdem ist eine Druckausgabe von einem Applet aus nicht möglich.

3 Java Grundlagen

Java ist eine *case-sensitive* Sprache, d.h. es wird zwischen Groß- und Kleinschreibung unterschieden. Java ist streng typisiert; Variable müssen vor ihrer Benutzung vereinbart werden. Variable können zudem noch mit Zugriffsrechten und Modifikatoren versehen werden. Auf deren Bedeutung wird im einzelnen im Kapitel *Klassenkonzept von Java* näher eingegangen. Anweisungen werden in Java durch das Semikolon `;` voneinander getrennt.

3.1 Kommentare

In Java gibt es drei Arten von Kommentaren. Wird in einer Zeile auf das Zeichen `//` gestoßen, so ist der Rest der Zeile Kommentar. Kommentare, die sich über mehrere Zeilen erstrecken können, werden wie in C durch `/*` und `*/` eingeschlossen. Kommentare, die mit `/**` beginnen und auf `*/` enden können zur automatischen Generierung von Dokumentationen mit dem Programm `javadoc` genutzt werden. `javadoc` ist Teil des JDKs.

3.2 Literale

Es gibt in Java folgenden Integer Literale:

- dezimale Darstellung

`10`	vom Typ `int`
`10l` bzw. `10L`	vom Typ `long`

- oktale Darstellung

`010`	vom Typ `int`
`010l` bzw. `010L`	vom Typ `long`

- hexadezimale Darstellung

`0xa`	vom Typ `int`
`0xal` bzw. `0xaL`	vom Typ `long`

Fließkomma Literale:

- Standard Darstellung

`10.05`	vom Typ `double`
`10.05f` bzw. `10.05F`	vom Typ `float`

- Exponentialschreibweise

`1.05E1`	vom Typ `double`
`1.05E1f` bzw. `1.05E1F`	vom Typ `float`

Boolsche Literale:

- boolsche Literale mit den werten `true` und `false` vom Typ `boolean`.

Zeichen Literale

- Zeichen Literale sind vom Datentyp `char` und werden in einfache Anführungszeichen eingeschlossen. Ein Zeichen Literal besteht aus genau einem Zeichen oder einer Escape Sequenz.
  Beispiele: `'A'` `'a'` `'\t'` `'\nnn'` (n ist Oktalwert; Zeichen in oktaler Darstellung)`'` oder `\unnnn'` (n ist Hexadezimalwert; Zeichen in Unicode Darstellung)

Zeichenketten Literale

- Zeichenketten Literale sind eine Folge von Zeichen und/oder Escape Sequenzen, die in doppelte Anführungszeichen eingeschlossen sind. Zeichenketten Literale sind Instanzen der Klasse `String`.
  Beispiel: `"Hello, World"`

3.3 Namen

Namen von Methoden, Eigenschaften, Klassen usw. dürfen mit einem Buchstaben, einem Unterstrich _ oder dem Zeichen `$` beginnen. Ziffern sind nicht erlaubt. Außerdem dürfen die Namen nicht mit reservierten Wörtern übereinstimmen. Es wird zwischen Groß- und Kleinschreibung unterschieden.

3.4 Datentypen

In Java werden zwei Arten von Datentypen unterschieden. Die *einfachen Datentypen*, zu denen `boolean`, `char`, die Integer und Floating-Point Datentypen gehören, und *die Referenz-Datentypen* mit Klassen, Schnittstellen und Arrays.

Zusammengesetzte Datentypen wie `struct`, `records`, `unions` etc. sind in Java nicht vorgesehen. Derartige Typen sind durch das Klassenkonzept realisierbar. Beschäftigen wir uns zunächst mit den einfachen Datentypen:

Für jeden einfachen Datentyp gibt es auch eine Klassen, die diesen Typ repräsentiert. Dadurch erhält man Zugriff auf die jeweiligen Konstanten des Datentyps, z.B. `MIN_VALUE` oder `POSITIVE_INFINITY`. Außerdem ist dies notwendig, da einige Methoden nur Objekte als Parameter akzeptieren.

Integer-Datentypen

Schlüsselwort	Breite	Wertebereich	Klasse
`byte`	8 Bit	-128 bis 127	`Byte`
`short`	16 Bit	-32768 bis 32767	`Short`
`int`	32 Bit	-2147483648 bis 2147483647	`Int`
`long`	64 Bit	-9223372036854775808 bis 9223372036854775807	`Long`

Floating Point Datentypen:
Gleitkomma Zahlen werden in der Form $\pm m*2^e$ dargestellt. (ANSI/IEEE-754 Spezifikation)

Schlüsselwort	Breite	m	e	Klasse
`float`	32 Bit	0 bis 2^{24}	-149 bis 104	`Float`
`double`	64 Bit	0 bis 2^{53}	-1075 bis 970	`Double`

Character Datentyp:

Schlüsselwort	Breite	Wertebereich	Klasse
`char`	16 Bit	`\u0000` bis `\uffff`	`Character`

Boolean Datentyp

Schlüsselwort	Breite	Wertebereich	Klasse
`boolean`	1 Bit	`true` bzw. `false`	`Boolean`

Ab der Version 1.1 des JDK gibt es die Klasse `Bignum` mit der es möglich ist, numerische Werte mit praktisch beliebiger Genauigkeit darzustellen.

Java ist eine streng typisierte Sprache, d.h. Variablen müssen vor ihrer Verwendung vereinbart werden. Gleichzeitig kann eine Initialisierung stattfinden:

```
int i;
float Volume = 3.141;
```

Die zweite Klasse der Datentypen in Java sind die Referenz-Datentypen, d.h. bei der Erzeugung eines Objekts mit dem Operator `new` wird eine Referenz, ein Zeiger, auf dieses Objekt übergeben. Zeigerarithmetik wie in C ist allerdings in Java aus gutem Grund nicht möglich. (Felheranfälligkeit, Sicherheitsaspekte.)

Arrays:

In Arrays werden Objekte gleichen Typs gespeichert. Eine Deklaration sieht wie folgt aus:

```
int i[];
char[] c;
```

Die eckigen Klammern können nach dem Variablennamen oder nach dem Typbezeichner stehen und geben an, daß es sich bei der Variablen um ein Array handelt. Speicherplatz, d.h. wieviele Elemente das Array aufnehmen kann, wird über die Anweisung

```
i = new int[5];
```

zur Verfügung gestellt. Nach dem Operator `new` kann ein einfacher Datentyp oder eine Klasse stehen. Danach kann mit der Anweisung

```
i[2] = 18;
```

das dritte Element des Arrays auf den Wert `18` gesetzt werden. (Der Indexbereich von Arrays beginnt wie in C mit dem Wert `0`).

Die obigen Schritte können auch zusammenfassend abgekürzt werden. Deklaration und Allokation von Speicherplatz geschieht über die Anweisung

```
int[] i = new int[5];
```

Alle drei Schritte über

```
int[] i = {2,5,7,-3,18};
```

Java kennt auch mehrdimensionale Arrays:

```
float Matrix[][] = new float[4][5];
```

Das dritte Element der zweiten Zeile kann mit Hilfe der Anweisung

```
Matrix[1][2] = 4.5;
```

gesetzt werden.

Strings:

In Java wird ein String im Gegensatz zu C nicht durch ein Array vom Typ `char` dargestellt. Für Strings existieren eigene Klassen und zwar die Klasse `String` und `StringBuffer`. Im Gegensatz zu `StringBuffer` repräsentieren Objekte der Klasse `String` eine Zeichenkette, die nach ihrer Initialisierung nicht mehr verändert werden können.

`Beispiel:`

```
String myString;
StringBuffer myStringBuffer;

myString = new String("Hello, World");
myStringBuffer = new StringBuffer("Hello");
myStringBuffer = myStringBuffer + "World";
```

In beiden String Objekten befindet sich schließlich die Zeichenkette `"Hello,World"`. Durch die Angabe eines String wird automatisch ein String Objekt erzeugt (siehe letzte Anweisung im obigen Beispiel). Durch + werden zwei Strings miteinander verknüpft.

Wir behandeln die Datentypen Klassen und Interfaces in einem eigenen Kapitel, nachdem wir die Operatoren, Kontrollstrukturen und Methoden von Java kennengelernt haben.

3.5 Operatoren in Java

arithmetische Operatoren

Java kennt fünf verschiedene arithmische Operatoren:

Operator	Bedeutung	Beispiel
`+`	Addition	`5+8`
`-`	Subtraktion	`5-8`
`*`	Multiplikation	`5*8`
`/`	Division	`5/8`
`%`	Modulo	`5%8`

Inkrement- und Dekrementoperatoren:

Operator	Bedeutung	Beispiel
`++`	Inkrement um 1	`x++` bzw `++x`
`--`	Dekrement um 1	`x--` bzw `--x`

Der Operator `++` (bzw. `--`) kann sowohl als Präfix- wie auch als Postfixoperator verwendet werden. Der Unterschied wird durch folgendes Beispiel deutlich:

```
y=x++;
y=++x;
```

Im ersten Fall erhält `y` den Wert von `x`. Erst danach wird `x` um `1` inkrementiert. Im zweiten Fall wird `x` inkrementiert und das Resultat `y` zugewiesen.

Bit-Operatoren:

Operator	Bedeutung	Beispiel
`<<`	Verschiebung der Bits nach links	`x<<y`
`>>`	Verschiebung der Bits nach rechts	`x>>y`
`>>>`	Verschiebung der Bits nach rechts. (Das Vorzeichen wird mit verschoben.)	`x>>>y`
`&`	Bitweises UND	`x&y`
`|`	Bitweises ODER	`x|y`
`^`	Bitweises exclusives ODER	`x^y`
`~`	Bitweises Komplement	`~x`

Bei den Shift Operatoren gibt der zweite Operand an um wieviele Stellen die Bits des ersten Operanden nach links bzw. rechts verschoben werden. Das bitweise exclusive ODER liefert `1`, wenn genau eins der beteiligten Bits `1` ist und das andere `0`. Die Operanden sind Integer Datentypen.

Vergleichsoperatoren:

Operator	Bedeutung	Beispiel
`<`	kleiner als	`n<m`
`>`	größer als	`n>m`
`<=`	kleiner als oder gleich	`n<=m`
`>=`	größer als oder gleich	`n>=m`
`==`	gleich	`n==m`
`!=`	ungleich	`n!=m`

logische Operatoren:

Operator	Bedeutung	Beispiel
`&&`	logisches UND	`n && m`
`||`	logisches ODER	`n || m`
`!`	logisches NICHT	`!n`

Zuweisungsoperatoren:

Operator	Bedeutung	Beispiel
`=`	Zuweisung	`n=n*3`
`*=, += etc.`	Zusammengesetzte Zuweisungsoperatoren	`n+=m`

Bei den zusammengesetzten Operatoren ist `n+=m` eine verkürzte Schreibweise für `n = n+m`.

Weitere wichtige Operatoren sind `new`, für das Erzeugen neuer Objekte, und `instanceof` zum Test, ob ein Objekt eine Instanz eines bestimmten Klasse ist. (Beispiel: `"Hello, World" instanceof String` würde den wert `true` liefern.)

3.6 Kontrollstrukturen

Bedingte Ausführung:

An bedingten Anweisungen kennt Java die `if` und `switch` Anweisung.

```
if (Bedingung)
      Anweisung1;
else
      Anweisung2;
```

Hat die Bedingung den Wahrheitswert `true` wird die *`Anweisung1`* ausgeführt, ansonsten die *`Anweisung2`*. Der `else` Teil kann fehlen. Anstelle einer einzigen Anweisung kann auch ein Anweisungsblock stehen, der mit `{` und `}` geklammert wird. Folgendes Beispiel ermittelt das Maximum zweier Zahlen:

```
if (n>=m)
  max = n;
else
  max = m;
```

Natürlich können innerhalb von `if` Anweisungen weitere `if` Anweisungen stehen.

Die `switch` Anweisung erlaubt die bedingte Ausführung aufgrund des Werts eines Ausdrucks:

```
switch (Ausdruck) {
      case Konstante1:
            Anweisung1;
      case Konstante2:
            Anweisung2;
      default:
            Anweisung3;
```

```
}
```

Der in Klammern angegebene Ausdruck muß vom Typ `char`, `byte`, `short` oder `int` sein. In Abhängigkeit vom Wert des Ausdrucks wird zu der entsprechenden Konstanten verzweigt. Alle folgenden Anweisungen (auch die der weiteren `case` Fälle) werden ausgeführt. Will man dieses verhindern, so muß als (letzte) Anweisung in einem `case` Block das `break;` Statement stehen. Die `break` Anweisung bricht die Abarbeitung eines Blocks ab. Es wird mit den Anweisungen nach der `switch` Anweisung fortgefahren. Die `default:` Anweisung ist optional. Hier können Anweisungen angegeben werden, die dann ausgeführt werden, falls keine der `case` Fälle zutreffen.

Das folgende Programmfragment untersucht, welcher Operator in der Variablen `op` gespeichert ist.

```
char op='+';

switch (op) {
  case '+':
    System.out.println("Das Pluszeichen");
    break;
  case '-':
    System.out.println("Das Minuszeichen");
    break;
  case '*':
    System.out.println("Das Multiplikationsszeichen");
    break;
  case '/':
    System.out.println("Das Divisionszeichen");
    break;
  default:
    System.out.println("Unbekannter Operator");
}
```

Schleifen:

Schleifen dienen zur wiederholten Ausführung bestimmter Programmteile. In Abhängigkeit einer Bedingung wird festgelegt wie oft die Ausführung wiederholt werden soll. Die erste Möglichkeit bietet die `while` Schleife:

```
while (Bedingung)
      Anweisung;
```

Die `while` Schleife wird solange wiederholt, wie die Bedingung `true` ist. Statt einer einzelnen Anweisung kann natürlich auch ein Block von Anweisungen stehen. Im Normalfall wird innerhalb des Blocks die Bedingung derart verändert, daß sie einmal den Wert `false` erhält und somit mit den Anweisungen nach der `while` Schleife fortgefahren werden kann.

Innerhalb der `while` Schleife (und auch anderen Blöcken) können die Anweisungen `break;` oder `continue;` (häufig in Zusammenhang mit der `if` Anweisung) stehen. `break` verläßt die Schleife und `continue` fährt direkt mit dem nächsten Schleifendurchlauf fort.

Im folgenden Beispiel werden alle ungeraden Zahlen zwischen 1 und 10 ausgegeben:

```
int n=1;

System.out.println("Ungerade Zahlen zwischen 1 und 10:");
while (n <= 10) {
  if (n%2 != 0)
    System.out.println(n);
  n = n+1;
}
```

Ähnlich wie die `while` Schleife arbeitet die `do-while` Schleife, nur mit dem Unterschied, daß die Bedingung nicht zu Beginn, sondern erst am Ende des Schleifendurchlaufs getestet wird. D.h. `do-while` Schleifen werden mindestens einmal durchlaufen.

```
do
      Anweisung;
while (Bedingung);
```

Die `for` Schleife unterscheidet sich von der `while` Schleife insofern, daß Initialisierung, Bedingung und Inkrement mit im Schleifenkopf durchgeführt werden:

```
for (Initialisierung; Bedingung; Inkrement)
      Anweisung;
```

Das obige Beispiel müßte folgendermaßen mit der `for` Schleife programmiert werden.

```
System.out.println(„Ungerade Zahlen zwischen 1 und 10:“);
for (n=1; n <= 10; n=n+1) {
  if (n%2 != 0)
    System.out.println(n);
}
```

3.7 Methoden

Neben Variablen sind Methoden ein weiterer Bestandteil von Klassen. Logisch gesehen sind Methoden das, was in C *function*, in Pascal *procedure* oder *function* und in Fortran *subroutine* oder *function* ist.

Methoden dienen zur Strukturierung und Modularisierung von Programmen. Methoden können können mit Zugriffsrechten und Modifikatoren versehen werden, die im Kapitel *Klassenkonzept in Java* behandelt werden.

In den Beispielen haben wir schon häufiger Methoden aufgerufen wie z.B. die Methode `println()` des `out` Objekts. Methoden sind wie Variable an Klassen gebunden. Beginnen wir mit einem Beispiel:

```
import java.io.*;

public class method1 {

  public static void main(String args[]) {
    System.out.println ("Die ersten ungeraden Zahlen <= 10");
    oddNumbers();
  }

  static void oddNumbers() {
    int n;
    for (n=1; n<=10; n=n+1)
      if (n%2 != 0)
        System.out.println(n);
  }
}
```

Diese Klasse besteht aus zwei Methoden, der Methode `main()` und der Methode `oddNumbers()`. Vor dem Namen einer Methode, steht der Typ der Methode. Ist eine Methode als `void` deklariert, so liefert sie keinen Wert zurück. Nach dem Namen der Methode werden in Klammern sogenannte formale Parameter angegeben, mit Namen und Typ und voneinander durch Kommata getrennt. Besitzt eine Methode keine Parameter, werden nur die Klammern angegeben. (siehe `oddNumbers()`). Danach werden in geschweiften Klammern eine Menge von Anweisungen eingeschlossen, die bei Aufruf der Methode ausgeführt werden . Da die Methode `oddNumbers()` keinen Wert zurückliefert, sondern nur auf die Standardausgabe schreibt, kann sie in Zeile 3 der `main()` Methode einfach über ihren Namen aufgerufen werden.

In Java können in einer Klasse mehrere Methoden mit gleichem Namen definiert werden. Sie müssen sich allerdings in Typ und/oder Anzahl der Parameter voneinander unterscheiden. Man sagt, die Signatur der Methoden muß unterscheidbar sein. Bei Aufruf einer Methode sorgt das System dafür, daß die Methode mit der richtigen Signatur verwendet wird.

Häufig gibt es pro Klasse mehrere Konstruktoren. Das sind Methoden, die bei der Erzeugung eines Objekts mit Hilfe des `new` Operators automatisch aufgerufen werden. Diese Methoden haben alle den gleichen Namen. Mehr dazu im Kapitel *Klassenkonzept in Java*.

Methoden können rekursiv programmiert werden, d.h. eine Methode kann sich selber aufrufen. Dieses Konzept wird oftmals in Zusammenhang mit bestimmten Datenstrukturen, wie verkettete Listen (siehe auch Programmbeispiel im Anhang) oder (binäre) Bäume verwendet. Wir geben hier das Standardbeispiel der Berechnung der Fakultät einer natürlichen Zahl an.

```
  static long facul (int n) {

    if (n == 0)
      return 1;
    else
      return n*facul(n-1);
  }
```

Mit Hilfe der `return` Anweisung liefert diese Methode einen Wert vom Typ `long` zurück. Im Programm könnte diese Methode z.B. über

```
m = facul(4);
```

aufgerufen werden.

3.8 Ein- und Ausgabe

In Java erfolgt die gesamte Eingabe über sogenannte *Streams*. Das sind Datenkanäle, über die meistens `byte` Folgen übertragen werden. Wir beschäftigen uns in diesem Abschnitt mit der Eingabe über den Standard-InputStream (meistens die Tastatur), der Ausgabe über den Standard-OutputStream (meistens der Bildschirm), mit der Ein- und Ausgabe in Dateien (dafür sind die Klassen `FileInputStream` und `FileOutputStream` zuständig), sowie mit der Eingabe von Daten über die Kommandozeile. Java kennt weitere Streams z.B. für Pipes oder Filter.

Eingabe über die Kommandozeile:

Wir hatten gesehen, daß jede Java Applikation eine `main()` Methode besitzt, die automatisch beim Start des Programms ausgeführt wird. Diese Methode wird folgendermaßen deklariert:

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

Die `main()` Methode besitzt als Parameter ein Array von Strings. Wenn nun beim Start des Programms nach dem Programmnamen weitere Werte angegeben werden, so werden diese Werte als Strings in `args[0]`, `args[1]`, `args[2]`, ... gespeichert. Wenn nun eine Java Applikation den Namen HelloWorld trägt, so würde über den Aufruf

```
java HelloWorld Hello World
```

in `args[0]= "Hello"` und in `args[1]= "World"` stehen. Da Arrays Objekte sind, kann z.B. über `args.length` die Anzahl der übergebenen Parameter festgestellt werden. `args[0].length` würde die Anzahl der Zeichen von `args[0]` liefern, da `args[0]` ein String Objekt ist.

```
import java.io.*;

public class method2 {

  public static void main(String args[]) {
    System.out.println ("Die ersten ungeraden Zahlen <= " + args[0]);
    oddNumbers(Integer.parseInt(args[0]));
  }

  static void oddNumbers(int m) {
    int n;
    for (n=1; n<=m; n=n+1)
      if (n%2 != 0)
        System.out.println(n);
  }
}
```

Über die Kommandozeile wird in diesem Beispiel eingegeben, bis zu welcher Zahl ungerade Zahlen berechnet werden sollen. Die Methode `Integer.parseInt()` wandelt eine Zahl als String in einen Integer Wert um.

Auch die Methode `oddNumbers()` wurde leicht modifiziert. In der alten Version war sie total statisch, d.h. sie war nur in der Lage, ungerade Zahlen zwischen 1 und 10 auszugeben. Durch die Einführung des formalen Parameters `m` ist die Einschränkung aufgehoben.

Durch den Befehl

```
java method2 10
```

würden die ungeraden Zahlen zwischen 1 und 10 ausgegeben werden.

Eingabe über die Standardeingabe (Tastatur):

Das folgende Beispiel verdeutlicht die Dateneingabe über die Tastatur. Wie schon eingangs beschrieben, werden über Datenkanäle meistens Folgen von Bytes übertragen. Die Methode `System.in.read()` liest ein Byte-Array von der Tastatur und speichert es in der Variablen `number`. In `rc` wird die Anzahl der eingelesenen Bytes gespeichert. Über `new String(number)` wird das Byte Array in einen String und dieser schließlich mit `Integer.parseInt(s)` in die geforderte Integer Zahl konvertiert.

Interessant ist in diesem Zusammenhang noch der `try` und `catch` Block. Viele Methoden in Java erzeugen sogenannte Ausnahmen, die der Programmierer behandeln muß bzw. sollte. Das geschieht mit dem Paar `try/catch`. Nähere Einzelheiten folgen dazu im Kapitel *Behandlung von Ausnahmen*.

```
import java.io.*;

public class method3 {
  public static void main (String args[]) {
    byte number[] = new byte [2];
    int rc;

    System.out.println("Gib zwei Ziffern ein: ");
    try {
      rc = System.in.read(number);
    }
    catch (IOException e) {
      System.out.println(e.getMessage());
    }

    String s = new String(number);
    s = s.trim();

    System.out.println("Die ersten ungeraden Zahlen <= " + s );
    oddNumbers(Integer.parseInt(s));
  }

  static void oddNumbers (int m) {
    int n;
    for (n=1; n<=m; n=n+1) {
      if (n%2 != 0)
        System.out.println(n);
    }
  }
}
```

Jede Ziffer wird in einem Byte gespeichert. Wird eine mehrstellige Zahl angegeben, werden nur die ersten beiden Ziffern berücksichtigt.

Eingabe aus einer Datei:

Das folgende Beispiel liest aus einer Datei Daten ein. Über die Anweisung `new FileInputStream("test.dat")` wird die Datei `test.dat` an einen Input Stream gebunden und durch das Objekt `fin` repräsentiert. Mit der Anweisung `fin.read()` wird wie im obigen Beispiel ein Array von Bytes aus der Datei gelesen. Alles andere wurde unverändert übernommen.

```
import java.io.*;

public class method4 {

  public static void main(String args[]) {
    byte number[] = new byte[2];
    int rc;

    try {
        FileInputStream fin = new FileInputStream("test.dat");
        rc = fin.read(number);
        }
    catch (IOException e){
        System.out.println(e.getMessage());
    }


    System.out.println ("Die ersten ungeraden Zahlen <= " );
    String s = new String(number);
    s = s.trim();

    System.out.println(s);
    oddNumbers(Integer.parseInt(s));
  }

  static void oddNumbers(int m) {
    int n;
    for (n=1; n<=m; n=n+1)
      if (n%2 != 0)
        System.out.println(n);
  }
}
```

Ausgabe auf die Standardausgabe (Bildschirm):

Die wohl zentrale Methode zur Ausgabe von Informationen auf die Standardausgabe, haben wir in Beispielen schon kennengelernt: `System.out.println()` bzw. `System.out.print()`, wenn man keinen Zeilenvorschub erreichen will. Beide Methoden sind mehrfach überladen zur Ausgabe von Objekten, Strings, Integer- , Floatwerten usw.

```
int i=10;
System.out.println("Der Wert von i beträgt: " + i);
```

Obige `println()` Methode gibt einen mit dem Konkatenationsoperator `+` verknüpften String aus . Die Variable `i` wird automatisch in einen String Wert konvertiert.

Ausgabe in eine Datei:

Folgendes Beispiel demonstriert die Ausgabe von Informationen in eine Datei:

```
import java.io.*;

public class method5 {

  public static void main(String args[]) {
    byte number[] = new byte[2];

    number[0]=1;
    number[1]=0;
    try {
        FileOutputStream fout = new FileOutputStream("test.out");
        fout.write(number);
        }
    catch (IOException e){
        System.out.println(e.getMessage());
    }
  }
```

Im Beispiel wird ein Array von Bytes in die Datei `test.out` geschrieben.

In den Klassen `DataInputStream` bzw. `DataOutputStream` stehen dem Programmierer einige „höhere“ Ein- bzw. Ausgabemethoden zur Verfügung. Über die Anweisungen

```
DataInputStream din = new DataInputStream(System.in);  bzw.
DataOutputStream dout = new DataOutputStream(System.out);
```

kann man den Standard Input Stream bzw. den Standard Output Stream an einen Daten Input Stream bzw. Daten Output Stream binden. Analog kann ein `FileInputStream` (`FileOutputStream`) an einen Daten Insput Steam (Daten Output Steam) gebunden werden. Die Anweisung

```
str = din.readLine();
```

liest eine gesamte Zeile ein und

```
dout.writeChars(str);
```

schreibt einen String in den `OutputStream`. Die Methode `readLine()` der Klasse `DataInputStream` ist eine alte Methode des JDK 1.0 und *deprecated*. Ab JDK 1.1 kann man benutzen:

```
FileInputStream fin = new FileInputStream(args[0]);
BufferedReader din = new BufferedReader(new InputStreamReader(fin));
str = din.readLine();
```

Der Zugriff auf das Dateisystem ist aus Sicherheitsgründen nur Java Applikationen erlaubt.

4 Die Behandlung von Ausnahmen

In allen Programmen können Fehler (*Errors*) bzw. Ausnahmen (*Exceptions*) auftreten., z.B. bei der Division durch 0, bei Zugriff auf ein Array-Element, das außerhalb des Indexbereichs liegt, beim Versuch von einer Datei zu lesen, die nicht existiert usw.

In herkömmlichen Sprachen ist der Programmierer dafür verantwortlich, mögliche Fehler zu berücksichtigen und entsprechend darauf zu reagieren. Häufig wird die Behandlung möglicher Fehlerquellen wegen des Programmieraufwands vernachlässigt. Berücksichtigt man hingegen alle Ausnahmen, so wird der Programmcode dadurch nicht unbedingt lesbarer.

In dem Bestreben, diesem Dilemma zu entkommen und sichere Programme zu schreiben, bietet Java die Behandlung von Ausnahmen an. Die Fehlerbehandlung besteht aus mehreren Blöcken: Im `try` Block befindet sich Programmcode, der eventuell einen Fehler auslösen könnte. Im `catch` Block befinden sich Anweisungen, wie auf den Fehler reagiert werden soll. Einem `try` Block können mehrere `catch` Blöcke folgen. Im abschließenden `finally` Block befinden sich Anweisungen, die immer ausgeführt werden, unabhängig davon, ob ein Fehler aufgetreten ist oder nicht. Der `finally` Block ist optional. Damit ergibt sich folgenden Grundstruktur:

```
try {
      // Programmcode, in dem Ausnahmen auftreten können
}
catch(Ausnahme e) {
      // Behandlung des Fehlers
}
finally {
      // Abschließende Fehlerbehandlung
}
```

Als *`Ausnahme`* ist der Fehler anzugeben, der vom Programmcode erzeugt wurde. Das kann z.B. eine `IOException` oder ganz allgemein eine `Exception` sein. Betrachten wir folgendes Beispiel:

```
try {
    FileOutputStream fout = new FileOutputStream("test.out");
    fout.write(number);
    }
catch (IOException e){
    System.out.println(e.getMessage());
    }
```

Beide Anweisungen des `try` Blocks können eine `IOException` erzeugen. Die erste Anweisung, falls die Datei `test.out` nicht gefunden wurde, die zweite, wenn beim Schreiben ein I/O Fehler aufgetreten ist. Jede Methode, die eine Ausnahme liefern könnte, muß mit einem `try` Block umschlossen werden.

Tritt bei der ersten Anweisung schon ein Fehler auf, so wird sofort der entsprechende `catch` Block ausgeführt. Soll die zweite Anweisung in jedem Fall auch ausgeführt werden, so ist sie in einem separaten `try/catch` Block zu setzen.

Ob eine Methode eine Ausnahme erzeugen kann und wenn ja, welche, kann in Referenzbeschreibungen nachgelesen werden. Die Beschreibung für obige `write()` Methode lautet z.B.

```
public void write(byte b[]) throws IOException
```

Ist der aufgetretende Fehler zu gravierend, Sollte im `catch` Block mit der Anweisung `System.exit()` das Programm beendet werden.

Betrachten wir als noch einmal das Beispiel. Wird diese Applikation ohne einen Kommandozeilen Parameter aufgerufen, so wird eine `ArrayIndexOutOfBoundsException` erzeugt und das Programm abgebrochen: In der `System.out.println()` Anweisung wurde unzulässiger Weise versucht, auf `args[0]` zuzugreifen.

Mit der Ausnahmebehandlung kann man diesen Fehler abfangen:

```
import java.io.*;

public class method2 {

  public static void main(String args[]) {
    try {
      System.out.println ("Die ersten ungeraden Zahlen <= " + args[0]);
    }
    catch(Exception e) {
      System.out.println(" Fehler"  + e.getMessage());
      System.exit(1);
    }
    oddNumbers(Integer.parseInt(args[0]));
  }

  static void oddNumbers(int m) {
    int n;
    for (n=1; n<=m; n=n+1)
      if (n%2 != 0)
        System.out.println(n);
  }
}
```

Im dem `catch` Block wird vorsorglich das Programm beendet.

Ausnahmen sind in Java Objekte. Ausgehend von bestehenden Ausnahmeklassen ist die Definition eigener Ausnahmeklassen möglich. Das folgende Beispiel zeigt die Definition einer Methode, die eine Ausnahme erzeugt. Es wird dabei Bezug auf die Ausnahmeklasse `Exception` genommen.

```
import java.io.*;

public class myException {
  public static void main (String args[]) {
    try {
      TestArgs(args);
    }
    catch (Exception e) {
      System.out.println("Fehler bei Kommandozeilenparametern: " + e.getMessage());
      System.out.println("usage: java myException <par1> <par2> ");
      System.exit(1);
    }
    System.out.println("Der erste Parameter : " + args[0]);
```

```
    System.out.println("Der zweite Parameter: " + args[1]);
  }

  public static void TestArgs(String str[]) throws Exception {
    if (str.length != 2)
      throw (new Exception("Anzahl Parameter != 2"));
  }
}
```

Die Methode `TestArgs()` ist in der Lage ein `Exception` Objekt zu erzeugen. Die `Exception` Klasse ist die Basisklasse für alle Ausnahmen. (Von der Klasse `Throwable` sind die Klassen `Error` und `Exception` abgeleitet.) Das neue `Exception` Objekt wird mit einem String erzeugt, der in dem `catch` Block über die Methode `getMessage()` abgefragt werden kann.

5 Klassenkonzept von Java

5.1 Klassen und abgeleitete Klassen

Klassen sind die wohl wichtigsten Komponenten in Java. Um ein Objekt von einer bestimmten Klasse erstellen zu können, muß zuerst einmal die Definition der Klasse vorliegen. Java ist eine streng objekt-orientierte Programmiersprache, d.h. alle Programmteile werden durch Objekte repräsentiert. Die Definition eine Klasse wird durch das Schlüsselwort `class` eingeleitet, gefolgt vom Namen der Klasse. Innerhalb von geschweiften Klammern werden die Variablen und Methoden der Klasse angegeben:

```
class myClass {
      // Variablendefinition
      int myInteger;

      // Methodendefinition
      void myMethod() {
            // Anweisungen von myMethod()
      }
}
```

Ein neues Objekt der Klasse `myClass` könnte über die Anweisung

```
myClass mCl = new myClass();
```

erzeugt werden. Danach könnte auf Variable und Methode des Objekts `mCl` über `mCl.myInteger` bzw. `mCl.myMethod()` zugegriffen werden.

Variable, die außerhalb von allen Methoden einer Klasse definiert werden, werden auch Instanzvariable genannt und sind somit allen Methoden der Klasse bekannt.

Methoden können überladen werden (Polymorphismus), wie wir bereits im Abschnitt *Methoden* gesehen haben. D.h. Eine Klasse kann mehrere Methoden gleichen Names haben, die sich aber in ihrer Signatur voneinander unterscheiden müssen. Das gilt insbesondere auch für die sogenannten *Konstruktoren* einer Klasse.

Der Konstruktor ist eine Methode einer Klasse, der den gleichen Namen wie die Klasse besitzt und keinen Wert zurückliefert. Konstruktoren werden bei der Erzeugung der Klasse automatisch aufgerufen und führen häufig Initialisierungen durch. Fehlt einer Klasse ein Konstruktor, so wird automatisch der Konstruktor der Oberklasse aufgerufen. Konstruktoren können natürlich Parameter besitzen.

Um die von einem Objekt belegten Ressourcen wieder freizugeben werden normalerweise *Destruktoren* verwendet. In Java geschieht die Freigabe automatisch über Garbage Collection.

Die *Vererbung* ist ein grundlegender Mechanismus von objektorientierten Programmier-sprachen. Das bedeutet, daß eine neue Klasse die Instanzvariablen und Methoden einer bereits bestehenden Klasse übernehmen und weitere eigene Instanzvariablen und Methoden

hinzufügen kann. Durch diese Art der Vererbung entsteht eine hierarchische Struktur. In Java sind alle Klassen von der Klasse `Object` abgeleitet. Mit anderen Worten bedeutet dies, daß alle Klassen in Java die Instanzvariablen und Methoden von `Object` kennen.

Um in Java eine neue Klasse von einer bereits bestehenden Klasse abzuleiten, muß das Schlüsselwort `extends` angegeben werden. Es darf immer nur von einer Klasse abgeleitet werden. Mehrfachvererbung wie in C++ ist in Java nicht möglich.

Weiterhin gibt es noch zwei spezielle Variable: `this` und `super`. `this` repräsentiert das aktuelle Objekt und `super` das Elternobjekt.

Doch geben wir nun endlich ein Beispiel, um die Dinge zu verdeutlichen:

```
import java.io.*;

public class MinMaxTest {
  public static void main(String args[]) {
    double[] a ={1,2,3,4,5,6,7,8};
    double[] b ={9,10};
    MinMax mm = new MinMax(0.0,0.0);
    extMinMax emm = new extMinMax();
    System.out.println("Test der Klasse MinMax ...");
    System.out.println("lastMin und lastMax: ");
    System.out.println(mm.lastMin + " und " + mm.lastMax);
    System.out.println("Min " + mm.Min(2.,4.) + " und Max " + mm.Max(2.,5.));

    System.out.println("Test der Klass extMinMax: ");
    System.out.println("lastMin und lastMax: ");
    System.out.println(emm.lastMin + " und " + emm.lastMax);
    System.out.println("Min " + emm.Min(a));
    System.out.println("Max " + emm.Max(a));
    System.out.println("Min " + emm.Min(b));
  }
}

class MinMax {
  double lastMin;
  double lastMax;

  MinMax() {
    this.lastMin = Double.NEGATIVE_INFINITY;
    this.lastMax = Double.POSITIVE_INFINITY;
  }

  MinMax(double lastMin, double lastMax) {
    this.lastMin = lastMin;
    this.lastMax = lastMax;
  }

  double Min(double a, double b) {
    if (a >= b)
      this.lastMin = b;
    else
      this.lastMin = a;
    return this.lastMin;
  }

  double Max(double a, double b) {
    if (a >= b)
      this.lastMax = a;
    else
      this.lastMax = b;
    return this.lastMax;
  }
}


class extMinMax extends MinMax {
  extMinMax() {
    super();
  }
```

```
  double Min(double[] a) {
    int i;
    double min=Double.POSITIVE_INFINITY;
    if (a.length == 2)
      return super.Min(a[0],a[1]);
    else {
      for (i=0; i <= a.length-1; i=i+1)
        if (a[i] <= min)
          min = a[i];
      super.lastMin = min;
      return min;
    }
  }

  double Max(double[] a) {
    int i;
    double max=Double.NEGATIVE_INFINITY;
    if (a.length == 2)
      return super.Max(a[0],a[1]);
    else {
      for (i=0; i <= a.length-1; i=i+1)
        if (a[i] >= max)
          max = a[i];
      super.lastMax = max;
      return max;
    }
  }
}
```

Das obige Programm erzeugt die folgende Ausgabe:

```
F:\usr\java>e:\jdk1.1.4\bin\java MinMaxTest
Test der Klasse MinMax ...
lastMin und lastMax:
0.0 und 0.0
Min 2.0 und Max 5.0
Test der Klass extMinMax:
lastMin und lastMax:
-Infinity und Infinity
Min 1.0
Max 8.0
Min 9.0
```

In dem Beispiel werden drei Klassen implementiert. `MinMaxTest` dient zum Testen der Klassen `MinMax` und `extMinMax`. `extMinMax` ist von `MinMax` abgeleitet und kann somit alle Instanzvariablen und Methoden von `MinMax` verwenden.

Die Klasse `MinMax` dient zur Berechnung des Minimums bzw. Maximums zweier Zahlen. In den Instanzvariablen `lastMin`, bzw. `lastMax` wird das Minimum resp. Maximum der letzten Berechnung gespeichert.

Die Klasse besitzt zwei Konstruktoren, einen ohne Parameter, durch den die Instanzvariable auf Standardwerte gesetzt werden und einen mit Parameter. Die Methoden `Min()` und `Max()` berechnen das Minimum und Maximum zweier Zahlen, speichern das Ergebnis in der entsprechenden Instanzvariablen und geben zusätzlich das Ergebnis aus.

Die Klasse `extMinMax` ist von `MinMax` mit Hilfe des Schlüsselwortes `extends` abgeleitet. Diese Klasse besitzt nur einen parameterlosen Konstruktor, der seinerseits über die `super()` Anweisung den parameterlosen Konstruktor der Oberklasse (also `MinMax()`) aufruft.

Die beiden Methoden berechnen das Minimum bzw. Maximum von einem Array von Zahlen. Hat das Array die Länge 2, so wird über `super.Min()` bzw. `super.Max()` die Methode der Oberklasse aufgerufen. Im anderen Fall wird das Feld Element für Element untersucht und auf diese Weise der minimale bzw. maximale Wert festgestellt.

5.2 Schnittstellen (Interfaces)

Eine Schnittstelle ist eine bestimmte Form einer Klasse. In Schnittstellen werden Methoden vorgeben, die noch keinen Programmcode enthalten. Es werden nur Namen und Parameter spezifiziert. Die Klasse, die diese Methoden der Schnittstelle verwenden möchte, muß sie schließlich selbst implementieren.

Die Definition einer Schnittstelle erfolgt über das Schlüsselwort `interface`, gefolgt vom seinem Namen. Eine Schnittstelle kann mit Hilfe von `extends` von bereits bestehenden Schnittstellen abgeleitet werden. Außer Methoden kann eine Schnittstelle auch Variable enthalten, die allerdings mit `final` als Konstante deklariert und initialisiert werden müssen.

Betrachten wir folgendes Beispiel:

```
interface myInterface extends Interface1, Interface2 {
  final float pi=3.14;

  public abstract float CalclulateCircleArea(double r);
    // r is radius
}
```

Die Schnittstelle `myInterface` ist von zwei weiteren Schnittstellen abgeleitet und stellt eine Konstante und eine Methodendeklaration zur Verfügung.

Das Einbinden dieser Schnittstelle in eine Klasse kann am besten an einem Beispiel verdeutlicht werden:

```
class circle implements myInterface {
  // Instanzvariablen der Klasse circle

  public float CalculateCircleArea(double r) {
     return (r*r/2) * myInterface.pi;
  }
  // weitere Methoden der Klasse circle
}
```

Um die Schnittstelle `myInterface` in die Klasse `circle` einzubinden, wird das Schlüsselwort `implements` verwendet. Es können hier mehrere Schnittstellen, durch Kommata getrennt, angegeben werden. In der Klasse `circle` wird die Methode `CalculateCircleArea()`

schließlich implementiert. Es können dabei auch die Konstanten der verwendeten Schnittstellen verwendet werden.

Da Java keine Mehrfachvererbung wie z.B. C++ kennt, werden Interfaces u.a. genutzt, um Mehrfachvererbungen nachzubilden. Außerdem werden Interfaces in Zusammenhang mit dem (im neuen JDK 1.1) Event-Handling (siehe auch Kapitel *Applets*) oder Threads benötigt.

5.3 Packages

Normalerweise wird in jeder Java Quellcode Datei nur eine Klasse implementiert. Der Name der Klasse muß dabei mit dem Namen der Quellcode Datei vor der Extension (`.java`) übereinstimmen. Der `javac` Compiler erzeugt daraus eine Byte-Code Datei (mit der Extension `.class`). Wird in einer Quellcode Datei nur eine Klasse als `public` deklariert, so dürfen in dieser Datei mehrere Klassen implementiert werden (siehe auch obiges Beispiel). Für jede Klasse wird eine separate `.class` Datei vom Compiler angelegt. Wieder muß der Name der `public` Klasse mit dem Namen der Datei vor der Extension übereinstimmen.

Pakete (*Packages*) bieten die Möglichkeit, mehrere Klassen unter einem Namen zusammen zu fassen. Das geschieht, indem zu Beginn der Quellcode Datei mit Hilfe des Schlüsselwortes `package` ein Paketname festgelegt wird. Der Paketname enthält die Information, wo die Klassen, die zu dem Paket gehören, in der Verzeichnisstruktur zu finden sind.

Beispiele:

```
package myUtils;

public class myClass {
  // Definition der Klasse myClass
}
```

```
package myUtils.myMath;

public class MinMax {
  // Definition der Klasse MinMax
}
```

Durch das erste Beispiel wird die Klasse `myClass` dem Paket `myUtils` hinzugefügt. Der Paketname gibt an, daß die Datei `myClass.class` unter WindowsNT z.B. im Verzeichnis `c:\jdk\classes\myUtils` zu finden ist. Ganz analog das zweite Beispiel: Die Klasse `MinMax` wird dem Paket `myUtils.myMath` hinzugefügt. Klassen die zu diesem Paket gehören, sind im Verzeichnis `c:\jdk\classes\myUtils\myMath` zu finden. Natürlich können viele Klassen z.B. zum Paket `myUtils` gehören. Dazu muß in jeder Quellcode Datei die Anweisung `package myUtils;` stehen.

Durch den Import von Paketen bzw. Klassen können diese in Programmen verwendet werden. In vielen Beispielen haben wird dieses schon gesehen. Durch die Anweisung

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

werden z.B. alle Klassen des Pakets `java.io` importiert. Ganz gezielt können einzelne Klassen eines Pakets durch

```
import myUtils.myMath.MinMax;
```

importiert werden.

5.4 Zugriffsrechte und Modifikatoren

Mit Zugriffsrechten ist in diesem Zusammenhang die Sichtbarkeit von Instanzvariablen und Methoden *für andere Klassen* gemeint. Das Schlüsselwort, das das jeweilige Zugriffsrecht festlegt, wird von der Definition von Variabeln oder Methoden angegeben. Z.B

```
public int i;                 oder
protected void GiveAnswer()
```

Folgende Tabelle gibt eine Übersicht über Zugriffsrechte auf Variable oder Methoden:

Schlüsselwort	eigene Klasse	Elternklasse	Package-Klassen	alle Klassen
`private`	ja	nein	nein	nein
`private protected`	ja	ja	nein	nein
`protected`	ja	ja	ja	nein
`public`	ja	ja	ja	ja
`friendly`	ja	nein	ja	nein

Wird keines der Schlüsselworte angegeben, so wird als Defaultwert `friendly` angenommen. Außerdem können noch Klassen und Interfaces das Zugriffsrecht `public` erhalten. Dadurch werden sie auch außerhalb des Pakets sichtbar. Voreingestellt ist nur die Sichtbarkeit innerhalb des Pakets.

Zusätzlich zu den Zugriffsrechten gibt es für Klassen, Schnittstellen, Methoden und Variablen noch bestimmte Modifikatoren. Modifikatoren legen weitere Eigenschaften fest. Den `static` Modifikator haben wir für Variable und Methoden schon kennengelernt.

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

Folgende Tabelle listet die möglichen Modifikatoren für Klassen, Schnittstellen, Methoden und Variablen auf. Im Anschluß daran werden sie näher erläutert.

	abstract	final	static	native	synchronized	transient	volatile
Klassen	x	x					
Schnittstellen	x						
Methoden	x	x	x	x	x		
Variable		x	x			x	x

`abstract`:
Der Modifikator `abstract` gibt an, daß eine Klasse noch nicht vollständig implementiert ist. Das bedeutet, daß diese Klasse nicht instanziiert, sondern nur weitervererbt werden kann. Ist eine Methode `abstract` deklariert, so muß sie in einer `abtract` deklarierten Klassen enthalten sein. Eine abstrakte Methode ist noch nicht fertig implementiert, sondern besteht nur aus der Deklaration. Von einer abstrakten Klasse muß eine neue Klasse abgeleitet werden, in der alle abstrakten Methoden implementiert werden. Erst dann ist diese Klasse instanziierbar.
Eine Schnittstelle ist automatisch `abstract`.

`final`:
Für Klassen bedeutet der Modifikator `final`, daß keine Unterklassen von dieser Klasse abgeleitet werden können. Als `final` deklarierte Methoden können in einer Unterklasse nicht überschrieben werden. Wird eine Variable als `final` deklariert, so wird sie als Konstante behandelt und muß bei ihrer Deklaration initialisiert werden.

`static`:
Wird eine Methode oder Variable `static` deklariert, so erreicht man, daß für alle Instanzen einer Klasse nur jeweils eine Methode bzw. Variable angelegt wird. Das bedeutet, daß alle Instanzen auf die gleiche Variable (Klassenvariable) bzw. Methode zugreifen. Um statische Variable bzw. Methoden verwenden zu können, muß nicht eine Instanz dieser Klasse erzeugt werden (siehe `main()` Methode).

`native`:
Eine mit `native` gekennzeichnete Methode besteht nur aus ihrer Deklaration. Die Implementierung liegt in einer anderen Sprache z.B C vor.

`sychronized`:
Damit beim Zugriff von Methoden aus verschiedenen Threads auf Objekte oder Variable keine unerwünschte Effekte entstehen, müssen diese Methoden synchronisiert werden. Das wird durch den Modifikator `synchronized` erreicht.

`transient`:
Als `transient` deklarierte Variable werden nicht automatisch (durch Objektserialisierung) in eine Datei mit abgespeichert.

`volatile`:
Als `volatile` deklarierte Variable haben für Threads eine besondere Bedeutung. Die Daten dieser Variablen werden vor jedem Lese- bzw. Schreibvorgang aktualisiert.

6 Applets

Bisher wurden Java Applikationen betrachtet, die keine graphische Benutzeroberfläche (*Graphical User Interface, GUI*) besaßen. Obwohl Java hier mit modernen Konzepten aufwartet (Klassenkonzept, Threads, Netzwerkfähigkeit, Plattformunabhängigkeit, Behandlung von Ausnahmen ...) ist die *Logik* derartiger Programme seit langem bekannt. Der Programmfluß läuft vom Beginn bis zum Ende eines Programms.

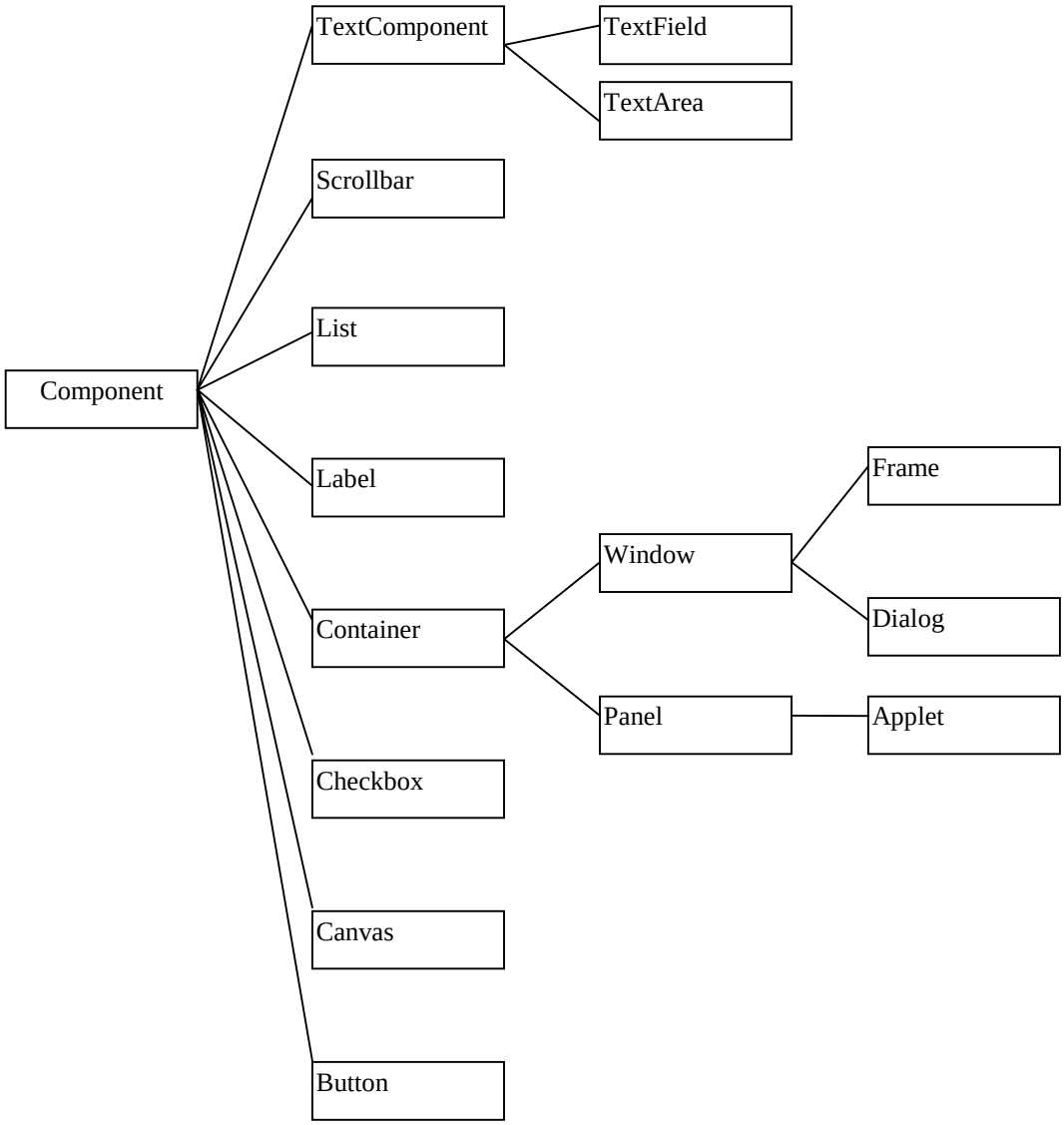
Programme mit graphischen Oberflächen und auch Realtime-Systeme reagieren auf Ereignisse (*Events*). Das System sammelt Ereignisse, wie z.B. Mausklick, Tastendruck, usw. und die Applikation verarbeitet sie und reagiert mit bestimmten Aktionen.

Java kennt zwei verschiedene Arten von Programmen mit graphischen Benutzeroberflächen: *Applets* und *GUI Applikationen*. In der folgenden Tabelle sind wesentliche Unterschiede zusammengestellt:

Applet	GUI Applikation
• Wird von einem java-fähigen WWW Browser ausgeführt.	• Wird von der JVM z.B. vom Programm `java` ausgeführt.
• Kein Aufbau eines Hauptfensters erforderlich, da bereits durch den Browser vorhanden.	• Ein Hauptfenster muß aufgebaut werden.
• Ein Applet durchläuft während seines „Lebens“ verschiedene „Meilensteine“	• Trifft nicht zu.
• Ein Applet kann nur zum dem Rechner eine Netzwerkverbindung aufbauen, von dem es geladen wurde.	• Trifft nicht zu.
• Ein Applet kann nicht auf das lokale Filesystem zugreifen	• Trifft nicht zu.
• Ein Applet kann keine Druckausgabe durchführen	• Trifft nicht zu.

Bei der Programmierung der graphischen Elemente und beim Event-Handling sind keine grundlegenden Unterschiede zwischen Applets und Applikationen zu nennen.

Graphische Applikationen werden mit dem *Abstract Window Toolkit* (*AWT*) programmiert. Durch die `import java.awt.*;` Anweisung werden die benötigten Klassen sichtbar gemacht. Zum weiteren Verständnis ist die Klassenhierarchie des AWT ganz hilfreich:



Eine graphische Anwendung besteht aus einer Anzahl von Komponenten (*Components*). Sowohl ein Fenster, wie auch Schaltflächen fallen unter diesen Begriff. Es gibt gewisse Komponenten, *Container* genannt, die die Fähigkeit haben, andere Komponenten zu enthalten. Damit die Komponenten sinnvoll am Bildschirm angeordnet werden können, ist ihnen ein Layoutmanager zugeordnet.

Container gliedern sich in *Panels* und *Windows*. Windows sind eigenständige Fenster mit Rahmen und Titel, in das z.B. eine GUI Applikation dargestellt werden kann. Ein Panel ist einem Window äquivalent, kann aber nicht eigenständig dargestellt werden. D.h ein Panel muß in einem Applet oder Window liegen.

6.1 Ein einfaches Java Applet

Folgendes Beispiel demonstriert ein einfaches Applet mit zugehöriger HTML Datei:

HTML Datei:

```
<html>
<head>
</head>
<body>
Applet: Applet1
<p>
<applet code="applet1.class" width=400 height=300>
</applet>
</body>
</html>
```

Applet:

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

public class applet1 extends Applet {
  public void init() {
    System.out.println ("Hello, World");
    System.out.println("Methode init() wird aufgerufen");
  }

  public void paint (Graphics g) {
    g.drawString("Hello, World", 50, 50);
    System.out.println("Methode paint() wird aufgerufen");
  }

  public void start() {
    System.out.println ("Methode start() wird aufgerufen");
  }

  public void stop() {
    System.out.println ("Methode stop() wird aufgerufen");
  }

  public void destroy() {
    System.out.println ("Methode destroy() wird aufgerufen");
  }

}
```

Über die HTML Datei wird mit Hilfe des `<applet>` Tags das Applet geladen und ausgeführt. Im allgemeinen hat der `<applet>` Tag folgende Form:

```
<applet code="AppetClassDatei" width=Pixel height=Pixel codebase=URL >
<param name="Parametername" value="Parameterwert">
<param name="Parametername" value="Parameterwert">
          ...
</applet>
```

Im einleitenden Applet-Tag sind noch weitere Attribute möglich, die wir hier aber nicht weiter betrachten. Der `code` Parameter gibt den Namen der Bytecode Datei an. `width` und `height`

bestimmen die Größe des Applets in Pixel. Mit `codebase` kann man ein Verzeichnis angeben, in dem sich das zu ladende Applet befindet.

Dem Applet können Parameter über den `<param ...>` Tag mitgegeben werden. Mit dem Einlesen von Parameter beschäftigen wir uns im nächsten Abschnitt. Abgeschlossen wird die gesamte Sequenz über `</applet>`.

Kommen wir nun zur Erläuterung des Applets:
Zunächst werden durch die `import` Anweisung notwendige Klassen importiert. Alle Applets sind Unterklassen der `Applet` Klasse. Die Methoden `init()`, `start()`, `stop()` und `destroy()` sind Methoden der Klasse `Applet` und können, wie hier geschehen, überschrieben werden. Diese Methoden werden auch *Meilensteine* im Leben eines Applets genannt. Die Methode `paint()` gehört zur Klasse `Component`.

Die Ausgaben über `System.out.println()` sind Kontrollausgaben und werden in nicht in das Fenster des Browsers geschrieben, in dem das Applet dargestellt wird. Die Kontrollausgaben verdeutlichen, welche Methode gerade ausgeführt wird. Unter Netscape hat man die Möglichtkeit über `Options->Show Java Console` ein weiteres Fester einzublenden, im dem Ausgaben auf die Standardausgabe dargestellt werden. Sollen Informationen im Applet dargestellt werden, so sind andere Methoden, wie z.B. `paint()` zu wählen.

Alle Methoden dieses Applets sind überschrieben und werden beim Eintreffen bestimmter Ereignisse automatisch aufgerufen:

- `init()`.
  Die `init()` Methode wird aufgerufen, wenn das Applet geladen wird. In dieser Methode sollten allgemeine Initialisierungen vorgenommen werden.
- `start()`.
  Die `start()` Methode wird nach der `init()` Methode aufgerufen. Hier können allgemeine Funktionen des Applets, wie z.B. das Abspielen einer Sound Datei aufgerufen werden.
- `stop()`.
  Die `stop()` Methode wird aufgerufen, wenn die HTML Seite verlassen wird oder der Browser minimiert wird. In ihr sollten Funktionen, die sich noch in Ausführung befinden, beendet werden.
- `destroy()`.
  Die `destroy()` Methode wird aufgerufen, wenn der Browser beendet wird.

Die Methode `paint()` dient zur Darstellung eines graphischen Kontextes (z.B. Texte oder Zeichnungen, Bilder). In unserem Beispiel wird durch diese Methode der String „Hello, World“ an die Stelle (50,50) in Pixel geschrieben. `paint()` wird automatisch aufgerufen bei der ersten Darstellung des Applets oder wenn das Browser Fenster neu gezeichnet werden muß.

Hat sich der graphische Kontext verändert, so kann mit `repaint()` eine Neudarstellung erzwungen werden.

6.2 Parameter und Applets

In vielen Fällen ist es wichtig, daß ein Applet konfiguriert werden kann. Angenommen, es soll ein Applet programmiert werden, daß einen Lauftext darstellt. Der Lauftext hat verschiedene Eigenschaften:

- welcher Text soll dargestellt werden
- Größe des Textes
- Font des Textes
- Ort der Darstellung im Applet
- Farbe des Textes usw.

Es ist sicherlich kein guter Stil, diese Attribute statisch ins Applet zu programmieren. Denn dann müßte für jeden Lauftext ein verändertes Applet geschrieben werden. Sinnvoller ist es, diese Attribute von außen über Parameter einzulesen.

```
<html>
<head>
</head>
<body>
Applet: Applet3
<p>
<applet code="applet3.class" width=400 height=300>
<param name="Title" value="Dies ist eine Überschrift">
</applet>
</body>
</html>
```

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

public class applet3 extends Applet {
  String str;

  public void init() {
    str = getParameter("Title");
    if (str == null)
       str = "Default Title";
  }

  public void paint (Graphics g) {
    g.drawString(str, 50, 50);
  }

  public String getAppletInfo() {
    String ainfo="Beispiel Applet3, W.Meyer ";
    return ainfo;
  }

  public String[][] getParameterInfo() {
    String pinfo[][]={{"Titel","String","Titel des Applets"}};
    return pinfo;
  }

}
```

Die Methode `getParameter()` holt den Wert des Parameters mit dem Namen Titel aus der HTML Datei und speichert ihn in die Variable `str`. (Wird kein Parameter angegeben, so wird ein Default Wert angenommen.) Die `paint()` Methode stellt den Text dar.

Die beiden anderen Methoden dienen zur Information und sollten implementiert werden. Damit ein Anwender, der ein fremdes Applet in eine HTML Seite einbindet, weiß, welche Parameter dieses Applet verwendet, gibt es die Methode `getParameterInfo()`. Diese Methode liefert ein 2-dimensionales Array von Strings zurück. Das erste Element enthält dabei den Namen des Parameters, das zweite den Typ und das dritte eine kurze Beschreibung. Die Methode `getAppletInfo()` liefert eine String, der z.B. den Namen des Applets, Version, Copyrightvermerke usw. enthalten sollte. Beide Methoden gehören zur Klasse `Applet` und werden im Beispiel überschrieben.

Diese Informationen über das Applet bzw. die Parameter kann z.B. im `appletviewer` abfragt werden, indem man die Option `Applet->Info...` wählt.

6.3 Komponenten, Container und Layoutmanager

Container sind in der Lage, mehrere Komponenten, wie z.B. Checkboxen, Buttons usw., aufzunehmen. Einem Container kann ein Layoutmanager zugewiesen sein, der für die Anordnung der einzelnen Komponenten im Container (und damit auf dem Bildschirm) verantwortlich ist.

Einfache Komponenten

Beschäftigen wir uns zunächst mit einfachen Komponenten. Es werden nur jeweils wenige Methoden der einzelnen Klassen kurz erwähnt. Eine komplette Aufstellung findet man in der Referenz.

- Button:
  Durch die Klasse `Button` wird eine einfache Schaltfläche repräsentiert. Der Button kann mit einem Text versehen sein, der entweder im Konstruktor angegeben oder mit der Methode `setLabel()` gesetzt werden kann. Mit Hilfe von `getLabel()` kann der Text abgefragt werden. Die Größe des Buttons richtet sich zum einen nach dem darzustellenden Text. Sie kann aber auch durch Layoutmanager beeinflußt werden.

  Beispiel:
  ```
  Button myButton = new Button("Press Me");
  myButton.setLabel("Ok");
  ```

- Canvas:
  Die Klasse `Canvas` repräsentiert eine Fläche, auf der man zeichnen kann oder z.B. Gif Bilder darstellen kann.

  Beispiel:
  ```
  Canvas myCanvas = new Canvas();
  ```

- Checkbox:
  Eine Objekt der Klasse `Checkbox` stellt sich als Kästchen mit (evtl.) nachfolgendem Text dar. Durch einen Mausklick kann das Kästchen markiert werden. Mehrere Checkboxen lassen sich durch einen besonderen Konstruktor zu einer `CheckboxGroup` zusammenfassen, in der nur genau ein Kästchen markiert sein kann. Es stehen eine Reihe von Methoden zur Verfügung. Mit `getState()` kann man z.B. den Zustand der Checkbox abfragen.

  Beispiel:
  ```
  Checkbox myCheckbox = new Checkbox("Show");
  bool = myCheckbox.getState();
  ```

- Choice:
  Bei einem Objekt der Klasse `Choice` handelt es sich um ein Pop-Up Menü mit der Darstellung des ausgewählten Elements. Über die Methode `addItem()` wird dem Menü ein Element hinzugefügt. Die Methode `getSelectedItem()` liefert das ausgewählte Element.

  Beispiel:
  ```
  Choice myChoice = new Choice();
  myChoice.addItem("Element1");
  myChoice.addItem("Element2");
  ```

- Label:
  Die `Klasse` Label stellt ein einzeiliges nicht edierbares Textfeld dar.

  Beispiel:
  ```
  Label myLabel = new Label("Hallo, World");
  ```

- List:
  Die Klasse `List` repräsentiert ein mehrzeiliges Textfeld, das nicht edierbar ist. Mit der Methode `addItem()` fügt man der Liste ein neues Element hinzu. Im Konstruktor kann man die Anzahl der Zeilen eines `List` Objekts angeben. Übersteigt die Anzahl der Elemente die Anzahl der Zeilen, so werden automatisch Scrollbars angelegt. Mit der Maus können Listenelemente ausgewählt werden. Mehrfachauswahl ist zulässig. Außerdem muß nicht unbedingt eine Element ausgewählt sein. Die Methode `getSelectedItems()` liefert die ausgewählten Elemente.

  Beispiel:
  ```
  List myList = new List();
  myList.addItem("Hallo");
  myList.addItem("World");
  ```

- Scrollbar:
  Die Klasse `Scrollbar` stellt Schiebeleisten dar. Im Konstruktor kann u.a. die Orientierung der Schiebeleiste (vertikal, horizontal) und der minimale bzw. maximale Wert angegeben werden, der durch Verschiebung der Leiste erreicht werden kann. Mit Hilfe der Methode `getValue()` kann der momentane Wert der Schiebeleiste abgefragt werden.

  Beispiel:
  ```
  Scrollbar myScrollbar = new Scrollbar();
  n = myScrollbar.getValue();
  ```

- TextField:
  In einem Objekt der Klasse `TextField` kann der Benutzer eine Textzeile eingeben. Mit den Methoden `getText()` bzw. `setText()` kann ein Text gelesen bzw. in das Textfeld gesetzt werden.

  Beispiel:
  ```
  TextField myTextField = new TextField("Default Text");
  myTextField.setText("Hallo, World");
  ```

- TextArea:
  Eine Objekt vom Typ `TextArea` ist eine mehrzeilige Text-Komponente. Die Methode `getText()` liefert den Text der TextArea.

  Beispiel:
  ```
  TextArea myTextArea = new TextArea("Hello,World",10,20);
  //Textarea with 10 rows and 20 colums and default text.
  Str = MyTextArea.getText();
  ```

Die Methoden `setText()` bzw. `getText()` sind in der Klasse `TextComponent` implementiert, die eine Oberklasse von `TextField` und `TextArea` ist.

Container

Die Klasse `Container` (und damit natürlich auch alle Unterklassen von `Container`) ermöglichen die Zusammenfassung mehrere Komponenten. Ihnen ist (standardmäßig) oder wird ein Layout zugewiesen, das die Darstellung der Komponenten am Bildschirm bestimmt. Typische Container sind z.B. Applets als Unterklasse von `Applet` oder `Frames` in GUI Applikationen.

Die Methode `add()` fügt eine Komponente einem Container hinzu. Durch die Methode `setLayout()` wird ein Layout eines Containers gesetzt. Mit Hilfe des Layoutmanagers werden die einzelnen Komponenten positioniert.

```
Button myButton = new Button("Ok");
FlowLayout myFlowLayout = new FlowLayout();
add(myButton);
setLayout(myFlowLayout);
```

Folgendes Beispiel erläutert einige einfache Komonenten und fügt sie über die Methode `add()` einem von der `Applet` abgeleiteten Klasse `ComponentDemo` hinzu. `ComponentDemo` ererbt damit auch alle Eigenschaften der Klasse `Container`:

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

public class ComponentDemo extends Applet {
```

```
  public void init () {

    // Einige Komponenten in JAVA ...

    Button myButton = new Button ("Ein Button");
    this.add(myButton);

    Checkbox myCheckBox = new Checkbox ("Eine CheckBox");
    this.add(myCheckBox);

    Choice myChoice = new Choice ();
    myChoice.addItem("Erstes Item");
    myChoice.addItem("Zeites Item");
    this.add(myChoice);

    Label myLabel = new Label("Ein Label");
    this.add (myLabel);

    List myList = new List ();
    myList.addItem ("Erstes Item");
    myList.addItem ("Zweites Item");
    this.add(myList);

    Scrollbar myScrollbar = new Scrollbar ();
    this.add (myScrollbar);

    TextField myTextField = new TextField ("Ein Text Feld");
    this.add(myTextField);

    TextArea myTextArea = new TextArea ("Eine Text Area");
    this.add (myTextArea);

  }
}
```

Das Applet stellt sich wie folgt auf dem Bildschirm dar:

Layoutmanager

Java kennt fünf verschiedene Layoutmanager: `FlowLayout`, `BorderLayout`, `CardLayout`, `GridLayout` und `GridBagLayout`. Applets ist standardmäßig das `FlowLayout` zugewiesen.

- FlowLayout:
  Das `FlowLayout` ordnet die einzelnen Komponenten in Reihen von links nach rechts an. Die Höhe der Reihen wird durch die Höhe der Komponenten bestimmt, die in ihr plaziert werden sollen. Paßt eine Komponente nicht mehr in eine Reihe, so wird auf die nächste Reihe übergegangen. Standardmäßig werden die Reihen zentriert dargestellt. Im Konstruktor kann man allerdings die Ausrichtung über die Konstanten `FlowLayout.RIGHT`, `FlowLayout.LEFT` oder `FlowLayout.CENTER` explizit angeben. Das FlowLayout wird häufig für Buttonleisten benutzt.

- GridLayout
  Das `GridLayout` teilt den Container in ein Gitter gleich großer Zellen. Im Konstruktor kann man die Anzahl der Reihen und Spalten, aber auch den Abstand zwischen den einzelnen Elementen angeben. Über die `add()` Methode werden die Komponenten von links nach rechs und von oben nach unten in die Zellen plaziert.

  Beispiel:
  ```
  GridLayout myGridLayout = new GridLayout(3,4);
  Button myButton = new Button("Submit");
  setLayout(myGridLayout);
  add(myButton);
  ```

- BorderLayout:
  Das `BorderLayout` teilt den Container in fünf Bereiche auf: `North`, `South`, `East`, `West` und `Center`. In der `add()` Methode muß zusätzlich der Ort angegeben werden, wo die Komponente dargestellt werden soll.

  Beispiel:
  ```
  BorderLayout myBorderLayout = new BorderLayout();
  Button myButton = new Button("Ok");
  setLayout(myBorderLayout);
  add("Center",myButton);
  ```

- CardLayout
  Das `CardLayout` dient dazu, mehrere Container wie bei einem Kartenstapel übereinander zulegen, zwischen denen man umschalten kann. Von den übereinander gelegten Containern ist immer nur einer (der obere) sichtbar. Über die `add()` Methode werden Komponenten dem Layout hinzugefügt, wobei zusätzlich ihnen noch ein interner Name zugewiesen wird. Dieser Name wird beim Umschalten zwischen den Karten benötigt.

  Beispiel:
  Das folgende Beispiel demonstriert u.a. das CardLayout.

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

public class applet5 extends Applet implements ActionListener {

  Panel myCards;

  public void init() {
    // 1. Karte
    Panel myCard1 = new Panel();
    Label myLabel1 = new Label("Karte Nr. 1");
    TextField myTextField1 = new TextField();
    myTextField1.setText("Text Feld der 1. Karte");
    myCard1.add(myLabel1);
    myCard1.add(myTextField1);

    // 2. Karte
    Panel myCard2 = new Panel();
    Label myLabel2 = new Label("Karte Nr. 2");
    Choice myChoice2 = new Choice();
    myChoice2.addItem("Element 1");
    myChoice2.addItem("Element 2");
    myChoice2.addItem("Element 3");
    myCard2.add(myLabel2);
    myCard2.add(myChoice2);

    // Zusammenführen der Karten ...
    Panel myCards = new Panel();
```

```
    CardLayout myCardLayout = new CardLayout();
    myCards.setLayout(myCardLayout);
    myCards.add("Karte1",myCard1);
    myCards.add("Karte2",myCard2);

    // Zwei Button mit deren Hilfe zwischen den
    // Karten hin- und hergeschaltet werden soll
    Panel myButtons = new Panel();
    Button myButton1 = new Button("Karte 1");
    Button myButton2 = new Button("Karte 2");
    myButtons.add(myButton1);
    myButtons.add(myButton2);
    myButton1.addActionListener(this);
    myButton2.addActionListener(this);

    // Layout des Applets
    BorderLayout myBorderLayout = new BorderLayout();
    setLayout(myBorderLayout);
    add("Center",myCards);
    add("South",myButtons);

  }

  public void actionPerformed(ActionEvent aevt) {
    if (aevt.getActionCommand() == "Karte 1")
      ((CardLayout)myCards.getLayout()).show(myCards,"Karte1");
    else
      ((CardLayout)myCards.getLayout()).show(myCards,"Karte2");
  }
}
```

Klickt man auf den Schaltknopf *Karte 1* wird die linke Abbildung angezeigt, klickt man auf den Schaltkopf *Karte 2* die rechte.

Zunächst wird für die erste Karte ein neues Panel erzeugt. Da ein Panel ein `Container` Objekt ist, kann es weitere Komponenten, in diesem Fall ein `Label` und ein `TextField` aufnehmen. Für die zweite Karte wird ganz analog vorgegangen. Dieser Container nimmt ein `Label` und ein `Choice` Objekt auf. Mit `add()` Methoden werden die Komponenten den entsprechenden Containern hinzugefügt. Beiden Containern ist standardmäßig das `FlowLayout` zugewiesen.

Danach wird ein weiteres Panel Objekt `myCards` erzeugt. Diesem Container wird das `CardLayout` zugewiesen. Mit einer speziellen Form der `add()` Methode werden die beiden Karten dem Objekt `myCards` hinzugefügt. Der String Parameter dieser Methode wird zum Umschalten zwischen den einzelnen Karten benötigt.

Im dritten Schritt wird ein weiteres Panel Objekt erzeugt, daß zwei Schaltknöpfe aufnehmen soll. Auch diesem Container ist das `FlowLayout` zugewiesen.

Im letzten Schritt wird schließlich das Layout des Applets (in diesem Fall `BorderLayout`) bestimmt. Das Objekt `myCards` wird in die Mitte und das Objekt `myButtons` unten im Applet aufgenommen.

Die anderen Teile des Applets sind für das Event-Handling zuständig, mit dem wir uns ausführlicher im nächsten Abschniit beschäftigen.

- GridBagLayout:
  Das `GrigBagLayout` ist der mächtigste aber auch komplizierteste Layoutmanager. Er arbeitet wie das `GridLayout` mit einem Gitter gleich großer Zellen. Es ist hier aber möglich, daß eine Komponente mehrere Zellen einnimmt. Außerdem kann die Lage der Komponenten festgelegt werden.
  Mit Hilfe der (Hilfs)Klasse `GridBagConstraints` werden Variable gesetzt, die die Lage einer Komponente kontrollieren. Mit der Methode `setConstraints()` werden sie einer Komponente zugewiesen, bevor diese Komponete mit `add()` einem Container zugewiesen wird.

6.4 Event-Handling

Bisher haben wir einige Möglichkeiten kennengelernt, Komponenten, wie Schaltknöpfe und Textfelder, auf dem Bildschirm zu positionieren. Das sieht zwar ganz nett aus, hat aber noch keinerlei Funktion. Ziel ist es, daß z.B. durch einen Klick auf einen Schaltknopf eine bestimmte Aktion ausgeführt wird. Das kann eine Funktion sein, die mit dem Schaltknopf verbunden ist. Dies wird durch Event-Handling realisiert.

Event-Handling ist von grundlegender Bedeutung für Programme mit graphischen Oberflächen. Der Benutzer löst durch Aktionen (Mausklick, Tastendruck, Focus auf ein Fenster bringen usw.) bestimmte Ereignisse (*Events*) aus. Das Programm muß in der Lage sein, auf das Eintreffen bestimmter Ereignisse mit der Ausführung gewünschter Funktionen zu reagieren.

Das Event-Handling hat sich in Java mit der Version 1.1 grundlegend geändert. Die alte Methode des Event-Handlings wird zwar weiterhin noch unterstützt, der Compiler liefert bei der Übersetzung aber eine Warnung. Außerdem hält sich SUN bedeckt, wie lange die alte Methode noch unterstützt wird. Da beide Methoden im Moment in Java ProgrammeVerwendung finden, gehen wir im folgenden kurz darauf ein.

Die alte Methode des Event-Handlings:

Für jedes Ereignis wird eine Instanz der Klasse `Event` erzeugt. In diesem Objekt sind alle notwendigen Informationen über das aufgetretende Ereignis enthalten. Mit Hilfe von Event-Handler werden aus Event-Objekten Information extrahiert und anschließend eine damit verbundene Funktion ausgeführt. In der folgenden Tabelle sind einige Variable der Klasse `Event` aufgeführt, die von Event-Handler zur Verarbeitung genutzt werden. Beim Eintreten eines Ereignisses sind nur die Variable gesetzt, die bezüglich dieses Events einen Sinn machen.

Variablennamen	Typ	Bedeutung	bei welchem Event gesetzt
`id`	`int`	Typ des Events	immer
`target`	`Object`	Komponenten, die den Event erzeugt hat	immer
`arg`	`Object`	zusätzliche Information zum Event	bei Listen- Scrollbar- Button - Choice - Events
`key`	`int`	Konstante der verwendeten Tase	bei Tastatur-Events
`modifiers`	`int`	Konstante von CTRL, ALT, Shift	bei Tastatur-Events
`when`	`long`	Zeitstempel	bei Tastatur- und Maus-Events
`x,y`	`int`	x und y Koordinate in Pixelwerte	bei Tastatur- und Maus-Events
`clickCount`	`int`	Anzahl der Mausklicks	bei Maus-Events
`evt`	`Event`	nächster Event	

Beispiel:

```
Button myButton = new Button("Hi");
add(myButton);
```

Findet ein Mausklick auf den so definierten Schaltknopf statt, so wird ein Event-Object mit folgenden Werten erzeugt:

```
id = ACTION_EVENT
target = myButton
arg = "Hi"
x = <x Koordinate>
y = <y Koordinate>
```

Die restlichen Variablen sind in diesem Fall nicht definiert. Die Methode `handleEvent()` dient zur Behandlung der Events. Das bedeutet, daß durch `handleEvent()` festgelegt wird, wie auf das Eintreffen bestimmter Ereignisse reagiert werden soll. `handleEvent()` gehört zur Klasse `Component` und kann in der eigenen Klasse überschrieben werden. Die Methode liefert `true` zurück, falls der Event behandelt wurde. Ist der Rückgabewert `false` muß der Event von einer der Oberklassen behandelt werden. `handleEvent()` hat folgende Default-Implementierung:

```
public boolean handleEvent(Event evt) {
switch (evt.id) {
  case Event.MOUSE_ENTER:
    return mouseEnter(evt,evt.x,evt.y);
  case Event.MOUSE_EXIT:
    return mouseExit(evt, evt.x,evt.y);
  case Event.MOUSE_MOVE:
```

```
    return mouseMouve(evt, evt.x,evt.y);
  case Event.MOUSE_DOWN:
    return mouseDown(evt, evt.x,evt.y);
  case Event.MOUSE_DRAG:
    return mouseDrag(evt, evt.x,evt.y);
  case Event.MOUSE_UP:
    return mouseUp(evt, evt.x,evt.y);
  case Event.KEY_PRESS:
  case Event.KEY_ACTION:
    return keyDown(evt, evt.key);
  case Event.KEY_RELEASE:
  case Event.KEY_ACTION_RELEASE:
    return keyUp(evt, evt.key);
  case Event.ACTION_EVENT:
    return action(evt, evt.arg);
  case Event.GOT_FOCUS:
    return gotFocus(evt, evt.arg);
  case Event.LOST_FOCUS:
    return lostFocus(evt, evt.arg);
  }
return false;
}
```

Es gibt noch einigere weitere Events, die nicht automatisch von `handleEvent()` verarbeitet werden. Dafür muß die Methode `handleEvent()` überschrieben werden, indem der Event entweder in der Methode oder durch eine Hilfsmethode behandelt wird. Die weiteren Events sind:

Event ID	Bedeutung
`WINDOW_DESTROY`	Fenster Event
`WINDOW_EXPOSE`	Fenster Event
`WINDOW_ICONIFY`	Fenster Event
`WINDOW_DEICONIFY`	Fenster Event
`WINDOW_MOVED`	Fenster Event
`SCROLL_LINE_UP`	Scrollbar Event
`SCROLL_LINE_DOWN`	Scrollbar Event
`SCROLL_PAGE_UP`	Scrollbar Event
`SCROLL_PAGE_DOWN`	Scrollbar Event
`SCROLL_ABSOLUTE`	Scrollbar Event
`LIST_SELECT`	List Event
`LIST_DESELECT`	List Event
`LOAD_FILE`	File Event
`SAVE_FILE`	File Event

Beispiele:
In einer GUI Applikation wird im Regelfall ein neues Fenster erzeugt. Daher sollte der Event `WINDOW_DESTROY` behandelt werden, da sich sonst die Applikation nur mit Gewalt, d.h. `^C` beenden läßt:

```
public boolean handleEvent(Event evt) {
  if (evt.id = Event.WINDOW_DESTROY)
    System.exit(0);
  return false;
}
```

Das folgende Applet zeichnet eine Linie vom ersten zum zweiten Klickpunkt. Dazu wird die Methode `mouseDown()` überschreiben:

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

public class applet6 extends Applet  {

int startx,starty,endx,endy;
int numOfPoints=0;

  public void paint (Graphics g) {
    if (numOfPoints == 4) {
      g.drawLine(startx,starty,endx,endy);
      numOfPoints = 0;
    }
  }

  public boolean mouseDown(Event evt, int x, int y) {
    switch (numOfPoints) {
      case 0: {
        startx = x;
        starty = y;
        numOfPoints = 2;
        return true;
        }
      case 2: {
        endx = x;
        endy = y;
        numOfPoints = 4;
        repaint();
        return true;
        }
    }
    return false;
  }
}
```

Das folgende Beispiel zeigt ein Applet, das einen Schaltknopf mit der Aufschrift `Press Me` enthält. Durch Mausklick auf diesen Button ändert sich die Aufschrift zu `Thanks ...` Klickt man auf eine andere Stelle des Applets, so wird `Press Me again` angezeigt. Dazu ist es nötig, die Methoden `action()` und `mouseDown()` zu überschreiben.

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

public class applet7 extends Applet  {

  Button myButton;

  public void init() {
    myButton = new Button("Press Me          ");
    add(myButton);
  }

  public boolean action(Event evt, Object arg) {
    if (evt.target == myButton) {
      myButton.setLabel("Thanks ...");
      return true;
      }
    else
      return false;
   }

  public boolean mouseDown(Event evt, int x, int y) {
    myButton.setLabel("Press Me again ");
    return true;
  }
}
```

Das neue Event-Handling

Bei dem neuen Event-Handling handelt es sich um ein sogenanntes *delegations basiertes* Modell: In diesem Modell werden Events von Event Quellen (*sourcen*) erzeugt. Sogenannte Zuhörer (*listener*) werden registriert und entscheiden durch ihre Implementation über die Antwort auf den Event. Außerdem gibt es mehrere Event Klassen. Doch der Reihe nach ...

`Listener` sind Interface Klassen, deren Methoden in der eigenen Klasse implementiert werden. Für Buttons ist z.B. der `ActionListener` verantworlich. Wie bekannt, muß ein Interface bei der Klassendefinition angegeben werden:

```
public class myApplet extends Applet implements ActionListener
```

In der folgenden Tabelle sind Listener Interfaces und deren Methoden zusammengestellt:

Listener Interface	Methoden
`ActionListener`	`actionPerformed(ActionEvent e)`
`AdjustmentListener`	`adjustmentValueChanged(AdjustmentEvent e)`

`ComponentListener`	`componentHidden(ComponentEvent e)`
	`componentMoved(ComponentEvent e)`
	`componentResized(ComponentEvent e)`
	`componentShown(ComponentEvent e)`

`ContainerListener`	`componentAdded(ContainerEvent e)` `componentRemoved(ContainerEvent e)`
`FocusListener`	`focusGained(FocusEvent e)` `focusLost(FocusEvent e)`
`ItemListener`	`itemStateChanged(ItemEvent e)`

`KeyListener`	`keyPressed(KeyEvent e)`
	`keyReleased(KeyEvent e)`
	`keyTyped(KeyEvent e)`

`MouseListener`	`mouseClicked(MouseEvent e)`
	`mouseEntered(MouseEvent e)`
	`mouseExited(MouseEvent e)`
	`mousePressed(MouseEvent e)`
	`mouseReleased (MouseEvent e)`

`MouseMotionListener`	`mouseDragged(MouseEvent e)` `mouseMoved(MouseEvent e)`
`TextListener`	`textValueChanged(TextEvent e)`

`WindowListener`	`windowActivated(WindowEvent e)`
	`windowClosed(WindowEvent e)`
	`windowClosing(WindowEvent e)`
	`windowDeactivated(WindowEvent e)`
	`windowDeiconified(WindowEvent e)`
	`windowIconified(WindowEvent e)`
	`windowOpened(WindowEvent e)`

Damit für eine Komponente wie z.B. der Button, das richtige Listener Interface implementiert werden kann, muß klar sein, welchen Event diese Komponente erzeugt. Das ist in der folgenden Tablen zusammengestellt:

	Event Klassen, die die Komponente erzeugen kann										
Komponete	action	adjustment	component	container	focus	item	key	mouse	mouse motion	text	window
Button	x		x		x		x	x	x		
Canvas			x		x		x	x	x		
CheckBox			x		x	x	x	x	x		
CheckBoxMenuItem						x					
Choice			x		x	x	x	x	x		
Component			x		x		x	x	x		
Container			x	x	x		x	x	x		
Dialog			x	x	x		x	x	x		x
Frame			x	x	x		x	x	x		x
Label			x		x		x	x	x		
List	x		x		x	x	x	x	x		
MenuItem	x										
Panel			x	x	x		x	x	x		
Scrollbar		x	x		x		x	x	x		
ScrollPane			x	x	x		x	x	x		
TextArea			x		x		x	x	x	x	
TextComponent			x		x		x	x	x	x	
TextField	x		x		x		x	x	x	x	
Window			x	x	x		x	x	x		x

Eine Beschreibung der einzelnen Event Klassen würde an dieser Stelle zu weit führen. Nähere Hinweise findet man z.B. unter `http://java.sun.com/docs/books/tutorial/post1.0/ui/eventsandcomponents.html`.

Die Programmierung von Event-Handlern erfolgt in drei Schritten:

- Das `class` Statement wird um die zu implementierenden Event Interfaces erweitert:
```
public class myApplet extends Applet
implements MouseListener,ActionListener
```

- Registrierung des Listeners an Komponenten.
```
Button myButton = new Button("Hi");
myButton.addActionListener(this);
```

- Implementation des Listener Interfaces.
```
public void actionPerformed(ActionEvent e) {
      // weitere Code
```

in den beiden folgenden Beispielen ist das Event-Handling auf die neue Methode umgestellt worden:

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

public class applet9 extends Applet implements MouseListener {

int startx,starty,endx,endy;
int numOfPoints=0;

  public void init() {
    addMouseListener(this);
  }

  public void paint (Graphics g) {
```

```
    if (numOfPoints == 4) {
      g.drawLine(startx,starty,endx,endy);
      numOfPoints = 0;
    }
  }

  public void mouseClicked(MouseEvent evt) {
    switch (numOfPoints) {
      case 0: {
        startx = evt.getX();
        starty = evt.getY();
        numOfPoints = 2;
        break;
        }
      case 2: {
        endx = evt.getX();
        endy = evt.getY();
        numOfPoints = 4;
        repaint();
       }
    }
  }

  public void mouseEntered(MouseEvent evt) {
  }

  public void mouseExited(MouseEvent evt) {
  }

  public void mousePressed(MouseEvent evt) {
  }

  public void mouseReleased(MouseEvent evt) {
  }

}
```

In der `init()` Methode wird der `MouseListener` an die Komponente `applet9` gebunden. Mit den Methoden `getX()` bzw `getY()` wird die x bzw. y Koordinate (in Pixel) aus dem `MouseEvent` übergeben.

Da es sich um ein Interface handelt, müssen alle Methoden des Interfaces, auch wenn sie nicht benötigt werden, implementiert werden. Ansonsten gibt es eine Fehlermeldung.

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

public class applet10 extends Applet implements ActionListener, MouseListener {

  Button myButton;

  public void init() {
    myButton = new Button("Press Me          ");
    add(myButton);
    myButton.addActionListener(this);
    addMouseListener(this);
  }

  public void actionPerformed(ActionEvent evt) {
    myButton.setLabel("Thanks ...");
  }

  public void mouseClicked(MouseEvent evt) {
    myButton.setLabel("Press Me again...");
  }

  public void mouseEntered(MouseEvent evt) {
  }

  public void mouseExited(MouseEvent evt) {
  }

  public void mousePressed(MouseEvent evt) {
  }
```

```
  public void mouseReleased(MouseEvent evt) {
  }


}
```

In diesem Beispiel wird an einen Button der `ActionListener` gebunden und der `MouseListener` an die Komponente `applet10`.

7 Anhang

7.1 Weitere Programmbeispiele

In diesem Teil finden Sie einige vollständige Programmbeispiele. Das erste Beispiel zeigt, wie aus Applikationen heraus eine Druckausgabe des graphischen Kontexts erreicht werden kann. Diese Möglichkeit besteht erst ab der Version 1.1 des JDK.

Beispiel 1:

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

public class PrintJob1 extends Frame
                       implements WindowListener, ActionListener {

  Button myPrint = new Button("Print");
    // Über diesen Button soll die Printausgabe angeregt werden.
  PrintCanvas canvas;
    // Auszugebende Graphik

  public static void main (String argv []) {
      Frame myFrame = new PrintJob1();
      myFrame.setSize(200,300);
      myFrame.show();
    }

  public PrintJob1() {

    addWindowListener(this);
      // Registrierung des WindowListeners
    myPrint.addActionListener(this);
      // Registrierung des ActionListeners
    setTitle(this.getClass().getName());
    canvas = new PrintCanvas();
      // Erzeugung eines graphischen Kontextes
    add("Center", canvas);
    add("North",myPrint);

    pack();

  }

  public void actionPerformed(ActionEvent e) {
     System.out.println ("Button");
        java.awt.PrintJob pjob = this.getToolkit().getPrintJob(this,"Test",null);
           // Erzeugen eines PrintJob Objekts
          if (pjob != null) {
          Graphics pg = pjob.getGraphics();
           // Das graphische Objekt wird geholt ...
          if (pg != null) {
            canvas.print (pg);
              // ... und ausgegeben...
            pg.dispose();
              //  Ende der Print Seite
          }
          pjob.end();
            // Ende des Print Jobs
        }
  }

  public void windowClosing(WindowEvent e) {
```

```
    System.out.println("Ende ...");
    System.exit(0);
  }

   public void windowClosed(WindowEvent event) {
         }
   public void windowDeiconified(WindowEvent event) {
         }
   public void windowIconified(WindowEvent event) {
         }
   public void windowActivated(WindowEvent event) {
         }
   public void windowDeactivated(WindowEvent event) {
         }
   public void windowOpened(WindowEvent event) {
         }
}

class PrintCanvas extends Canvas {

  public void paint(Graphics g) {
        g.setColor(Color.black);
        g.setFont(new Font("SansSerif",Font.BOLD,16));
        g.drawString("Hallo, World",50,50);

       }
}
```

Das folgende Beispiel beschäftigt sich mit verketteten Listen und implementiert dazu zwei Methoden `PrintList()` zur Ausgabe der gesamten Liste und `AppendList()` zum Anhängen eines Objekts an das Ende der Liste. Die Listenelemente können beliebige Objekte vom Typ `ListObject` sein.

Da Java über keine Zeiger im herkömmlichen Sinn verfügt, wird die Verkettung über Objekte realisiert. Das ist möglich, da bei der Erzeugung von Objekten Referenzen auf diese Objekte übergeben werden.

Beispiel 2:

```
// WM, RZ Uni Os

public class linkedLists {

  public ListObject myListObject;

  public class ListObject {
    public int i;
    public ListObject nextListObject;

    public ListObject(int i) {
      this.i = i;
    }
  }

  public void PrintList(ListObject lo) {

    while (lo != null) {
      System.out.println(lo.i);
      lo = lo.nextListObject;
    }
  }

  public ListObject AppendList(ListObject lo, int n) {
    ListObject help;


   if (lo == null)
     lo = new ListObject(n);
   else {
```

```
     help = lo;
     while (lo.nextListObject != null)
       lo = lo.nextListObject;
     lo.nextListObject = new ListObject(n);
     lo = help;
    }
   return lo;
  }

  public void ListTest () {
    myListObject = AppendList(myListObject,2);
    myListObject = AppendList(myListObject,3);
    myListObject = AppendList(myListObject,4);
    myListObject = AppendList(myListObject,1);
    PrintList(myListObject);
  }

  public static void main (String argv []) {
    linkedLists mylinkedLists = new linkedLists();
    mylinkedLists.ListTest();
  }
}
```

Folgendes Beispiel ist aus [2] entnommen und demonstriert in hervorragender Weise die neuen Möglichkeiten des JDK 1.1. Es handelt sich um eine Applikation mit Menüpunkten und Pop-up Menü.

Beispiel 3:

```
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.awt.*;               // ScrollPane, PopupMenu, MenuShortcut, etc.
import java.awt.datatransfer.*;  // Clipboard, Transferable, DataFlavor, etc.
import java.awt.event.*;         // New event model.
import java.io.*;                // Object serialization streams.
import java.util.zip.*;          // Data compression/decompression streams.
import java.util.Vector;         // To store the scribble in.
import java.util.Properties;     // To store printing preferences in.

/**
 * This class places a Scribble component in a ScrollPane container,
 * puts the ScrollPane in a window, and adds a simple pulldown menu system.
 * The menu uses menu shortcuts.  Events are handled with anonymous classes.
 */
public class ScribbleFrame extends Frame {
  /** A very simple main() method for our program. */
  public static void main(String[] args) { new ScribbleFrame(); }

  /** Remember # of open windows so we can quit when last one is closed */
  protected static int num_windows = 0;

  /** Create a Frame, Menu, and ScrollPane for the scribble component */
  public ScribbleFrame() {
    super("ScribbleFrame");                  // Create the window.
    num_windows++;                           // Count it.

    ScrollPane pane = new ScrollPane();      // Create a ScrollPane.
    pane.setSize(300, 300);                  // Specify its size.
    this.add(pane, "Center");                // Add it to the frame.
    Scribble scribble;
    scribble = new Scribble(this, 500, 500); // Create a bigger scribble area.
    pane.add(scribble);                      // Add it to the ScrollPane.

    MenuBar menubar = new MenuBar();         // Create a menubar.
    this.setMenuBar(menubar);                // Add it to the frame.
    Menu file = new Menu("File");            // Create a File menu.
    menubar.add(file);                       // Add to menubar.
```

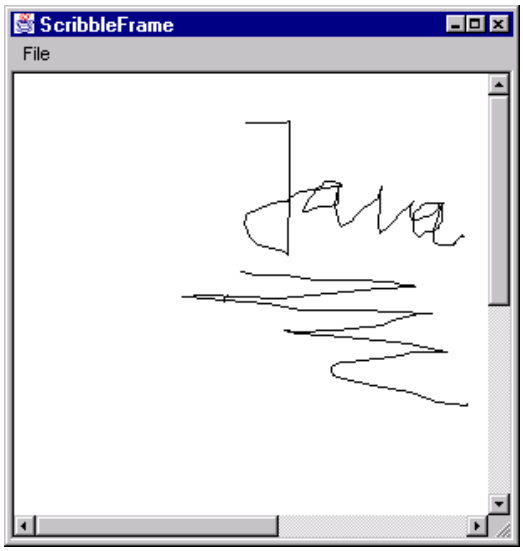
```
    // Create three menu items, with menu shortcuts, and add to the menu.
    MenuItem n, c, q;
    file.add(n = new MenuItem("New Window", new MenuShortcut(KeyEvent.VK_N)));
    file.add(c = new MenuItem("Close Window",new MenuShortcut(KeyEvent.VK_W)));
    file.addSeparator();                    // Put a separator in the menu
    file.add(q = new MenuItem("Quit", new MenuShortcut(KeyEvent.VK_Q)));

    // Create and register action listener objects for the three menu items.
    n.addActionListener(new ActionListener() {     // Open a new window
      public void actionPerformed(ActionEvent e) { new ScribbleFrame(); }
    });
    c.addActionListener(new ActionListener() {     // Close this window.
      public void actionPerformed(ActionEvent e) { close(); }
    });
    q.addActionListener(new ActionListener() {     // Quit the program.
      public void actionPerformed(ActionEvent e) { System.exit(0); }
    });

    // Another event listener, this one to handle window close requests.
    this.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { close(); }
    });

    // Set the window size and pop it up.
    this.pack();
    this.show();
  }

  /** Close a window.  If this is the last open window, just quit. */
  void close() {
    if (--num_windows == 0) System.exit(0);
    else this.dispose();
  }
}

/**
 * This class is a custom component that supports scribbling.  It also has
 * a popup menu that allows the scribble color to be set and provides access
 * to printing, cut-and-paste, and file loading and saving facilities.
 * Note that it extends Component rather than Canvas, making it "lightweight."
 */
class Scribble extends Component implements ActionListener {
  protected short last_x, last_y;               // Coordinates of last click.
  protected Vector lines = new Vector(256,256);  // Store the scribbles.
  protected Color current_color = Color.black;   // Current drawing color.
  protected int width, height;                   // The preferred size.
  protected PopupMenu popup;                     // The popup menu.
  protected Frame frame;                         // The frame we are within.

  /** This constructor requires a Frame and a desired size */
  public Scribble(Frame frame, int width, int height) {
    this.frame = frame;
    this.width = width;
    this.height = height;

    // We handle scribbling with low-level events, so we must specify
    // which events we are interested in.
    this.enableEvents(AWTEvent.MOUSE_EVENT_MASK);
    this.enableEvents(AWTEvent.MOUSE_MOTION_EVENT_MASK);

    // Create the popup menu using a loop.  Note the separation of menu
    // "action command" string from menu label.  Good for internationalization.
    String[] labels = new String[] {
      "Clear", "Print", "Save", "Load", "Cut", "Copy", "Paste" };
    String[] commands = new String[] {
      "clear", "print", "save", "load", "cut", "copy", "paste" };
    popup = new PopupMenu();                    // Create the menu
    for(int i = 0; i < labels.length; i++) {
      MenuItem mi = new MenuItem(labels[i]);    // Create a menu item.
      mi.setActionCommand(commands[i]);         // Set its action command.
      mi.addActionListener(this);               // And its action listener.
      popup.add(mi);                            // Add item to the popup menu.
    }
    Menu colors = new Menu("Color");            // Create a submenu.
    popup.add(colors);                          // And add it to the popup.
    String[] colornames = new String[] { "Black", "Red", "Green", "Blue"};
    for(int i = 0; i < colornames.length; i++) {
      MenuItem mi = new MenuItem(colornames[i]);  // Create the submenu items
      mi.setActionCommand(colornames[i]);         // in the same way.
      mi.addActionListener(this);
      colors.add(mi);
    }
```

```
  this.add(popup);
}

// Finally, register the popup menu with the component it appears over
/** Specifies big the component would like to be.  It always returns the
 *  preferred size passed to the Scribble() constructor */
public Dimension getPreferredSize() { return new Dimension(width, height); }

/** This is the ActionListener method invoked by the popup menu items */
public void actionPerformed(ActionEvent event) {
  // Get the "action command" of the event, and dispatch based on that.
  // This method calls a lot of the interesting methods in this class.
  String command = event.getActionCommand();
  if (command.equals("clear")) clear();
  else if (command.equals("print")) print();
  else if (command.equals("save")) save();
  else if (command.equals("load")) load();
  else if (command.equals("cut")) cut();
  else if (command.equals("copy")) copy();
  else if (command.equals("paste")) paste();
  else if (command.equals("Black")) current_color = Color.black;
  else if (command.equals("Red")) current_color = Color.red;
  else if (command.equals("Green")) current_color = Color.green;
  else if (command.equals("Blue")) current_color = Color.blue;
}

/** Draw all the saved lines of the scribble, in the appropriate colors */
public void paint(Graphics g) {
  for(int i = 0; i < lines.size(); i++) {
    Line l = (Line)lines.elementAt(i);
    g.setColor(l.color);
    g.drawLine(l.x1, l.y1, l.x2, l.y2);
  }
}

/**
 * This is the low-level event-handling method called on mouse events
 * that do not involve mouse motion.  Note the use of isPopupTrigger()
 * to check for the platform-dependent popup menu posting event, and of
 * the show() method to make the popup visible.  If the menu is not posted,
 * then this method saves the coordinates of a mouse click or invokes
 * the superclass method.
 */
public void processMouseEvent(MouseEvent e) {
  if (e.isPopupTrigger())                              // If popup trigger,
    popup.show(this, e.getX(), e.getY());              // pop up the menu.
  else if (e.getID() == MouseEvent.MOUSE_PRESSED) {
    last_x = (short)e.getX(); last_y = (short)e.getY(); // Save position.
  }
  else super.processMouseEvent(e);  // Pass other event types on.
}

/**
 * This method is called for mouse motion events.  It adds a line to the
 * scribble, on screen, and in the saved representation
 */
public void processMouseMotionEvent(MouseEvent e) {
  if (e.getID() == MouseEvent.MOUSE_DRAGGED) {
    Graphics g = getGraphics();                     // Object to draw with.
    g.setColor(current_color);                      // Set the current color.
    g.drawLine(last_x, last_y, e.getX(), e.getY()); // Draw this line
    lines.addElement(new Line(last_x, last_y,       // and save it, too.
                              (short) e.getX(), (short)e.getY(),
                              current_color));
    last_x = (short) e.getX();  // Remember current mouse coordinates.
    last_y = (short) e.getY();
  }
  else super.processMouseMotionEvent(e);  // Important!
}

/** Clear the scribble.  Invoked by popup menu */
void clear() {
  lines.removeAllElements();   // Throw out the saved scribble
  repaint();                   // and redraw everything.
}

/** Print out the scribble.  Invoked by popup menu. */
void print() {
  // Obtain a PrintJob object.  This posts a Print dialog.
  // printprefs (created below) stores user printing preferences.
  Toolkit toolkit = this.getToolkit();
  PrintJob job = toolkit.getPrintJob(frame, "Scribble", printprefs);
```

```
  if (job == null) return;

  // Get a Graphics object for the first page of output.
  // If the user clicked Cancel in the print dialog, then do nothing.
  Graphics page = job.getGraphics();

  // Check the size of the scribble component and of the page.
  Dimension size = this.getSize();
  Dimension pagesize = job.getPageDimension();

  // Center the output on the page.  Otherwise it would be
  // be scrunched up in the upper-left corner of the page.
  page.translate((pagesize.width - size.width)/2,
                 (pagesize.height - size.height)/2);

  // Draw a border around the output area, so it looks neat.
  page.drawRect(-1, -1, size.width+1, size.height+1);

  // Set a clipping region so our scribbles don't go outside the border.
  // On-screen this clipping happens automatically, but not on paper.
  page.setClip(0, 0, size.width, size.height);

  // Print this Scribble component.  By default this will just call paint().
  // This method is named print(), too, but that is just coincidence.
  this.print(page);

  // Finish up printing.
  page.dispose();   // End the page--send it to the printer.
  job.end();        // End the print job.
}

/** This Properties object stores the user print dialog settings. */
private static Properties printprefs = new Properties();

/**
 * The DataFlavor used for our particular type of cut-and-paste data.
 * This one will transfer data in the form of a serialized Vector object.
 * Note that in Java 1.1.1, this works intra-application, but not between
 * applications.  Java 1.1.1 inter-application data transfer is limited to
 * the pre-defined string and text data flavors.
 */
public static final DataFlavor dataFlavor =
    new DataFlavor(Vector.class, "ScribbleVectorOfLines");

/**
 * Copy the current scribble and store it in a SimpleSelection object
 * (defined below).  Then put that object on the clipboard for pasting.
 */
public void copy() {
  // Get system clipboard
  Clipboard c = this.getToolkit().getSystemClipboard();
  // Copy and save the scribble in a Transferable object
  SimpleSelection s = new SimpleSelection(lines.clone(), dataFlavor);
  // Put that object on the clipboard
  c.setContents(s, s);
}

/** Cut is just like a copy, except we erase the scribble afterwards */
public void cut() { copy(); clear();  }

/**
 * Ask for the Transferable contents of the system clipboard, then ask that
 * object for the scribble data it represents.  If either step fails, beep!
 */
public void paste() {
  Clipboard c = this.getToolkit().getSystemClipboard();  // Get clipboard.
  Transferable t = c.getContents(this);                  // Get its contents.
  if (t == null) {              // If there is nothing to paste, beep.
    this.getToolkit().beep();
    return;
  }
  try {
    // Ask for clipboard contents to be converted to our data flavor.
    // This will throw an exception if our flavor is not supported.
    Vector newlines = (Vector) t.getTransferData(dataFlavor);
    // Add all those pasted lines to our scribble.
    for(int i = 0; i < newlines.size(); i++)
      lines.addElement(newlines.elementAt(i));
    // And redraw the whole thing
    repaint();
  }
  catch (UnsupportedFlavorException e) {
    this.getToolkit().beep();   // If clipboard has some other type of data
```

```
  catch (Exception e) {
    this.getToolkit().beep();   // Or if anything else goes wrong...
  }
} }

/**
 * This nested class implements the Transferable and ClipboardOwner
 * interfaces used in data transfer.  It is a simple class that remembers a
 * selected object and makes it available in only one specified flavor.
 */
static class SimpleSelection implements Transferable, ClipboardOwner {
  protected Object selection;    // The data to be transferred.
  protected DataFlavor flavor;   // The one data flavor supported.
  public SimpleSelection(Object selection, DataFlavor flavor) {
    this.selection = selection;  // Specify data.
    this.flavor = flavor;        // Specify flavor.
  }

  /** Return the list of supported flavors.  Just one in this case */
  public DataFlavor[] getTransferDataFlavors() {
    return new DataFlavor[] { flavor };
  }
  /** Check whether we support a specified flavor */
  public boolean isDataFlavorSupported(DataFlavor f) {
    return f.equals(flavor);
  }
  /** If the flavor is right, transfer the data (i.e. return it) */
  public Object getTransferData(DataFlavor f)
       throws UnsupportedFlavorException {
    if (f.equals(flavor)) return selection;
    else throw new UnsupportedFlavorException(f);
  }

  /** This is the ClipboardOwner method.  Called when the data is no
   *  longer on the clipboard.  In this case, we don't need to do much. */
  public void lostOwnership(Clipboard c, Transferable t) {
    selection = null;
  }
}

/**
 * Prompt the user for a filename, and save the scribble in that file.
 * Serialize the vector of lines with an ObjectOutputStream.
 * Compress the serialized objects with a GZIPOutputStream.
 * Write the compressed, serialized data to a file with a FileOutputStream.
 * Don't forget to flush and close the stream.
 */
public void save() {
  // Create a file dialog to query the user for a filename.
  FileDialog f = new FileDialog(frame, "Save Scribble", FileDialog.SAVE);
  f.show();                          // Display the dialog and block.
  String filename = f.getFile();   // Get the user's response
  if (filename != null) {          // If user didn't click "Cancel".
    try {
      // Create the necessary output streams to save the scribble.
      FileOutputStream fos = new FileOutputStream(filename); // Save to file
      GZIPOutputStream gzos = new GZIPOutputStream(fos);     // Compressed
      ObjectOutputStream out = new ObjectOutputStream(gzos); // Save objects
      out.writeObject(lines);      // Write the entire Vector of scribbles
      out.flush();                 // Always flush the output.
      out.close();                 // And close the stream.
    }
    // Print out exceptions.  We should really display them in a dialog...
    catch (IOException e) { System.out.println(e); }
  }
}

/**
 * Prompt for a filename, and load a scribble from that file.
 * Read compressed, serialized data with a FileInputStream.
 * Uncompress that data with a GZIPInputStream.
 * Deserialize the vector of lines with a ObjectInputStream.
 * Replace current data with new data, and redraw everything.
 */
public void load() {
  // Create a file dialog to query the user for a filename.
  FileDialog f = new FileDialog(frame, "Load Scribble", FileDialog.LOAD);
  f.show();                          // Display the dialog and block.
  String filename = f.getFile();    // Get the user's response
  if (filename != null) {           // If user didn't click "Cancel".
    try {
      // Create necessary input streams
```

```
        GZIPInputStream gzis = new GZIPInputStream(fis);     // Uncompress
        ObjectInputStream in = new ObjectInputStream(gzis);  // Read objects
        // Read in an object.  It should be a vector of scribbles
        Vector newlines = (Vector)in.readObject();
        in.close();                     // Close the stream.
        lines = newlines;               // Set the Vector of lines.
        repaint();                      // And redisplay the scribble.
      }
      // Print out exceptions.  We should really display them in a dialog...
      catch (Exception e) { System.out.println(e); }
    }
  }

  /** A class to store the coordinates and color of one scribbled line.
   *  The complete scribble is stored as a Vector of these objects */
  static class Line implements Serializable {
    public short x1, y1, x2, y2;
    public Color color;
    public Line(short x1, short y1, short x2, short y2, Color c) {
      this.x1 = x1; this.y1 = y1; this.x2 = x2; this.y2 = y2; this.color = c;
    }
  }
}
```



Folgendes Beispiel stellt noch einmal vereinfacht Pull-Down und Pop-Up Menüs dar.

Beispiel 4:

```
//WM, RZ Uni OS

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

public class Menue extends Frame implements ActionListener{
```

```
  public static void main (String args[]) {
    Menue myMenue = new Menue();
    myMenue.setSize(400,400);
    myMenue.show();
  }

  public Menue() {

    setTitle("Menue's und Pop-Up's");

    // Menue
    MenuBar myMenuBar = new MenuBar();
    Menu myFileMenu = new Menu("File");
    Menu myHelpMenu = new Menu("Help");

    MenuItem quit = new MenuItem("Quit");
    quit.setActionCommand("quit");
    quit.addActionListener(this);
```

```
    MenuItem help = new MenuItem("About");

    myFileMenu.add(quit);
    myHelpMenu.add(help);

    myMenuBar.add(myFileMenu);
    myMenuBar.add(myHelpMenu);
    this.setMenuBar(myMenuBar);

    ScrollPane pane = new ScrollPane();
    add(pane);
    PopUp myPopUpMenu = new PopUp();
    pane.add(myPopUpMenu);

  }

  public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand() == "quit")
      System.exit(0);
  }

}

class PopUp extends Component implements ActionListener {
  public PopupMenu myPopUp;
  public MenuItem popupItem;

  public PopUp () {

    this.enableEvents(AWTEvent.MOUSE_EVENT_MASK);

    myPopUp = new PopupMenu();
    popupItem = new MenuItem("Quit");
    popupItem.setActionCommand("quit");
    popupItem.addActionListener(this);
    myPopUp.add(popupItem);
    popupItem = new MenuItem("About");
    popupItem.addActionListener(this);
    myPopUp.add(popupItem);
    myPopUp.addSeparator();

    this.add(myPopUp);
  }

  public void processMouseEvent(MouseEvent e) {
    if (e.isPopupTrigger())
      myPopUp.show(this,e.getX(),e.getY());
    else
      super.processMouseEvent(e);
  }

  public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand() == "quit")
      System.exit(0);
  }
}
```

Folgendes Programm kann sowohl als Applet als auch als GUI Applikation verwendet werden. Es läd als Hintergrund ein `*.bmp` Bild. Beim Laden des Bildes muß berücksichtigt werden, ob das Programm als Applet oder Applikation gestartet wird.

Beispiel 5:

```
// Programm als Applet und GUI Applikation
// WM, RZ Uni OS


import java.awt.*;
import java.applet.*;
import java.net.*;
import java.awt.event.*;

public class ImageApplet extends Applet implements ActionListener{
  static Image myImage;
  Button readmeButton = new Button("Readme");
  Button quitButton = new Button("Quit");
  static boolean runasApplet = true;

  public static void main (String args []) {
    Frame myFrame = new Frame();
    Applet myApplet = new ImageApplet();
    runasApplet = false;
    myFrame.add(myApplet);
    Toolkit tk = myFrame.getToolkit();
    // Plattformabhängige Schnittstellen ...

    myImage = tk.getImage("autorun.gif");
    myApplet.init();
    myApplet.start();
    myFrame.pack();
    myFrame.setSize(600,500);
    myFrame.show();
  }

  public void init () {

    if (runasApplet) {
      URL codeBase = getCodeBase();
      myImage = getImage(codeBase,"autorun.gif");
    }

    add(readmeButton);
    add(quitButton);
    readmeButton.addActionListener(this);
    quitButton.addActionListener(this);
  }

  public void paint (Graphics g) {
    g.drawImage (myImage,10,10,this);
  }
```

```
  public void actionPerformed (ActionEvent e) {
    if (e.getActionCommand() == "Quit") {
      System.out.println("QUIT");
   //   stop();
   //   destroy();
   //   System.exit(0);
    }
    else if (e.getActionCommand() == "Readme") {
      System.out.println("README");
    }
  }
}
```

Das Programm leistet folgende Ausgabe:

Folgendes JAVA Programm führt ein externes Programm als Prozess aus.

Beispiel 6:

```
// Aufruf von externen Programmen in Java
// WM, RZ Uni OS

import java.io.*;
import java.lang.*;

public class RunExApp {
  public static void main (String argv []) {
    String outstring;
    Process p;
    Runtime rt;
    int rc;

    System.out.println("Es geht los ...");
    rt = Runtime.getRuntime();
```

```
    // Liefert ein Runtime Objekt der momentanen Plattform zurueck.

    System.out.println(rt.freeMemory());

    try {
        //  p = rt.exec("cmd.exe /c dir");
        //  p = rt.exec("notepad.exe");
        p = rt.exec(argv[0]);
        // Startet einen neuen Prozess und liefert ein Process Objekt zurueck

        InputStreamReader istr = new InputStreamReader(p.getInputStream());
        // Ueber getImputStream() leitet man die Ausgabe des Prozesses um

        BufferedReader bstr = new BufferedReader(istr);
        while ((outstring = bstr.readLine()) != null) {
          System.out.println(outstring);
         }
    }
    catch (Exception e) {
      System.out.println("Fehler: " + e.getMessage());
    }
    System.out.println("Ende");
    }
}
```

8 Literatur

[1] Campione, M., Walrath, K.: The Java Tutoral, ACM
online: http://java.sun.com/docs/books/index.html

[2] Flanagean et. Al.: JAVA in a Nutshell, O'Reilly

[3] Glahn, K. u.a.: Java 1.1 mit Methode, C&L

[4] Harold, E.R.: Brewing Java: A Tutorial
http://sunsite.unc.edu/javafaq/javatutorial.html

[5] JAVA Unleshed, Sams

[6] Kühnel, R.: Die Java Fibel, Addison-Wesley

[7] Mintert, St.: Redevouz, in: IX, 7/96, S.110 ff

[8] Piemont, C.: Klappe,die 1.1ste, in: c't , 6/97, S. 364 ff

[9] Schreiner, A.T.: Programmieren mit Java, Skript

[10] Weber, J.: u.a.: Using JAVA, Que

[11] Yu, N.: Introduction to the AWT
http://java.cs.uni-magdeburg.de/dokus/AWT.Tutorial/AWT.Introduction.htm

Viele Beispiele finden Sie unter http://www.gamelan.com.