FU Berlin
WS 1998/99
Gerald Brose

Einführung in die Programmierung mit Java

Skript zum Blockkurs

Inhalt

1 EINFÜHRUNG 4

1.1 Grundbegriffe der Programmierung 4
1.1.1 Vom Algorithmus... 4
1.1.2 ...zum Programm 5
1.1.3 ...auf dem Rechner 5

1.2 Java-Einführung 7
1.2.1 einige Eigenschaften 7

2 GRUNDLEGENDE SPRACHKONZEPTE 9

2.1 Ausdrücke, Anweisungen, Kommentare 9

2.2 Klassen und Objekte 10
2.2.1 Objektzustand: Instanzvariablen 11
2.2.2 Objekterzeugung: Konstruktoren 12
2.2.3 this 12
2.2.4 Objektverhalten: Methoden 13

2.3 Kontrollstrukturen 17
2.3.1 Wiederholungsanweisungen (Schleifen) 17
2.3.2 bedingte Anweisungen (Verzweigung) 18
2.3.3 Sprunganweisungen 19

2.4 Typen in Java 20
2.4.1 Wozu Typen? 20
2.4.2 Basistypen und Operatoren: 21
2.4.3 Arrays 23
2.4.4 Strings (Zeichenketten) 24

2.5 Objektorientierung 27
2.5.1 Vererbung 27
2.5.2 Polymorphie (Vielgestaltigkeit) 27
2.5.3 Konstruktoren in Unterklassen 29
2.5.4 Überschreiben/Overriding von Methoden 29
2.5.5 Abstrakte Klassen 31
2.5.6 Interfaces (Schnittstellen) 32

2.6 Ausnahmebehandlung 33

2.7 Packages 34
2.7.1 Klassennamen und Packages 34
2.7.2 Packages und Zugriffsregeln 34

2.8 Ein-/Ausgabe 35
2.8.1 Streams 35

3 FORTGESCHRITTENE KONZEPTE 38

3.1 Inner Classes 38
3.1.1 Nested classes 38
3.1.2 Member classes 38

3.1.3 Local classes 39
3.1.4 Anonymous Classes 39

3.2 Programmiermethodik und Softwaretechnik 40
3.2.1 Komplexität bei der Softwareerstellung 40
3.2.2 Phasenmodell der SW-Erstellung 40
3.2.3 Funktionale Spezifikation 41
3.2.4 Spezifikation und Java 42

3.3 GUI-Programmierung 45
3.3.1 Klassen im AWT 45
3.3.2 Benutzereingaben und Events 49

3.4 Applets 53
3.4.1 Überblick 53
3.4.2 Ein HelloWorld-Applet 53
3.4.3 Einschränkungen für Applets 55

3.5 Threads 60
3.5.1 Nebenläufigkeit – eine Einführung 60
3.5.2 Nebenläufigkeit in Java: Threads 61
3.5.3 Sperrsynchronisation 63
3.5.4 Weitere Synchronisationsmöglichkeiten: Ereignisse, Join 66
3.5.5 Threads in Applets: Das QuickSort-Applet 66

1 Einführung

In diesem Kapitel wird eine allgemeine Einführung in die Programmierung und ein Kurzüberblick über die Programmiersprache Java gegeben.

1.1 Grundbegriffe der Programmierung

Dieser Abschnitt gibt einige einleitende Erläuterungen zu Grundbegriffen.

1.1.1 Vom Algorithmus...

- Ausgangspunkt ist ein gestelltes Problem, z.B. die Berechnung des größten gemeinsamen Teilers (ggT) zweier Zahlen

Def. Algorithmus:

Ein Algorithmus ist eine präzise, endliche Verarbeitungsvorschrift zur Lösung eines Problems.

- Eigenschaften von Algorithmen:
  - Die Beschreibung von Algorithmen ist *finit*, d.h. von endlicher Länge
  - Algorithmen sind *determiniert,* d.h. bei gleichen Parametern liefert jede Ausführung das gleiche Ergebnis.
  - Algorithmen *terminieren* i.d.R., d.h. sie enden nach endlich vielen Schritten (Ausnahme Steueralgorithemen, z.B. im Betriebssystem)
  - Algorithmen sind unabhängig vom konkreten Prozessor, der sie ausführt (Mensch mit Papier und Bleistift vs. Rechner)
- **Beispiel**: "euklidischer Algorithmus" zur Berechnung des ggT zweier Zahlen x und y.

  1. Falls x = y, dann ist ggT(x,y) = x
  2. Sonst:

     falls x > y: ersetze x durch x – y, weiter bei 1.

     Sonst: ersetze y durch y – x , weiter bei 1.
- Kein Algorithmus: "Der ggT von x und y ist die größte Zahl, durch die sich sowohl x als auch y teilen lassen."
- Beispiel: Einkaufen im Supermarkt:
  1. Gehe in den Supermarkt
  2. Nimm benötigte Artikel aus dem Regal
  3. Gehe zur Kasse

4. Erfrage Summe der Preise der Artikel
5. Zahle
6. Packe ein
7. Gehe aus dem Supermarkt

1.1.2 ...zum Programm

Def. Programm/Programmiersprache:
Ein Programm ist die Formulierung eines Algorithmus in einer konkreten Programmiersprache. Eine Programmiersprache ist ein durch einen Rechner interpretierbarer Formalismus mit eindeutig definierter Syntax und Semantik.

- **Die Syntax** ("Satzbau") einer Programmiersprachen beschreibt, welche Zeichenfolgen gültige Programme sind.

  "Er sieht das Mädchen mit dem Fernrohr" (gültig, jedoch zwei Analysen möglich)

- **Die Semantik** einer Programmiersprache definiert die genaue "Bedeutung" von Programmen in dieser Sprache fest.

  "Ich gebe Dir einen Korb"

  Syntaktisch korrekte sprachliche Äußerungen können mehrere Bedeutungen haben (s. Bsp.), in Programmiersprachen ist das nicht erwünscht.

1.1.3 ...auf dem Rechner

- Welche Sprachen verstehen Rechner? Was bedeutet "verstehen" hier?

  Der Rechner erkennt und bearbeitet nur Instruktionen in *Maschinensprache*, d.h. einem durch die konkrete Hardware definierten Satz von Binärcodes:

  `000100100111101010...` = „lade 4711 in die Speicheradresse 10232„

- "Assemblersprache": mnemonische Darstellung solcher Instruktionen

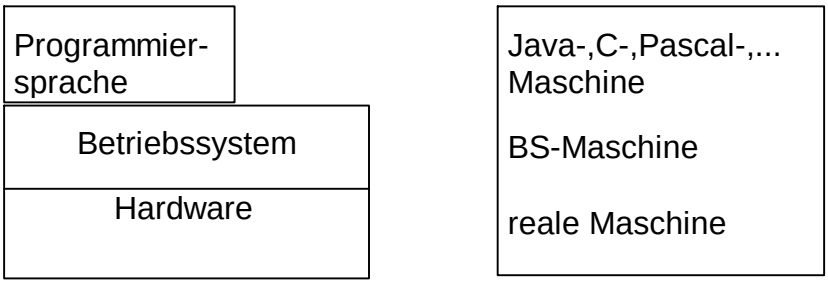
  Beispiel in einer hypothetischen Assemblersprache:

```
LABEL:   LDA   #3
         ADD   (7)
         STA    #4
         MOV   (27) #3
         JMP   LABEL
```

  „lade den Inhalt von Register 3 in das Akkumulatorregister, addiere 7 hinzu, speichere den Inhalt des Akkumulatorregisters in Register 4, speichere 27 in Register 3, setze die Programmausführung bei LABEL fort.“

⇒ Die Programmierung auf Maschinenebene ist umständlich, nicht problem- sondern maschinenbezogen, schwer lesbar und fehleranfällig – "aber die Programme sind schnell!"

- *Hochsprachen* sind abstrakter (problembezogener, "menschlicher"), kürzer, besser lesbar, weniger hardwareabhängig (im Idealfall portabel).
- Man verwendet häufig sog. *Schichtenarchitekturen*, um von rechnernahen Details zu abstrahieren - jede neue Schicht reichert die Funtkionalität der jeweils unterliegenden Schicht an und bietet so eine abstraktere Sicht:



- Eine virtuell "Programmiermaschine" wird durch einen Übersetzer und/oder Interpretierer auf einem konkretem Betriebssystem realisiert.
  - *Interpretierer*: ein Programm, das Programme liest und interpretiert, indem es direkt die entsprechenden Maschineninstruktionen ausführt.
  - *Übersetzer*: ein Programm, das ein „Quellprogramm“ in einer höheren Sprache in ein Maschinenprogramm übersetzt
- Allgemein werden Übersetzer bevorzugt, wenn hohe Effizienz benötigt wird. Interpretierer hingegen bieten höhere Flexibilität zur Laufzeit des Programms.

1.2 Java-Einführung

- Java ist eine *objektorientierte* (s. 2.5) Programmiersprache der Firma Sun Microsystems
- Zuerst 1995 offiziell vorgestellt, die aktuelle Version ist 1.1
- beispielloser Erfolg: noch nie hat eine neue Programmiersprache in so kurzer Zeit so starke Verbreitung gefunden

1.2.1 einige Eigenschaften

- *frei verfügbar*: Bibliotheken und Programmierwerkzeuge können kostenlos über das Netz von Sun heruntergeladen werden.
- erlaubt das *dynamische Laden* und Binden von Code, z.B. über das Netzwerk -> Applets
- *leicht zu lernen*: die Syntax ist stark an die bekannten und verbreiteten Programmiersprachen C/C++ angelehnt, so daß Programmierern solcher Sprachen der Umstieg leicht fällt. Außerdem ist Java an einigen Stellen gegenüber C/C++ stark vereinfacht
  -> weniger Programmierfehler, höhere Produktivität
- Integrierte Sprachkonzepte für Nebenläufigkeit -> *Threads*
- umfangreiche Bibliotheken für GUI- und verteilte Programmierung
- wird erst in ein maschinenunabhängiges Bytecode-Format übersetzt und dann interpretiert (*Java Virtual Machine*):

  Vorteil: Unabhängigkeit von realer HW (Portabilität): Java-Programme laufen ohne Änderung auf jedem Rechner, für den es eine JVM gibt („write once, run anywhere“).

  Beispiel:

```
public class HelloWorld {
     public static void main( String [] argv){
          System.out.println(„Hello World!“);
     }
}
```

  – übersetzen: `javac HelloWorld.java`

  – ausführen: `java HelloWorld`

  ggT in Java:

```
public class Primer {

     public static int ggT( int a, int b ){
          if( a == b )   // Achtung: == oder =
               return a;
```

```
          else if( a > b ) {
               System.out.println(„1. rekursiver Aufruf“);
               return ggT( a-b, b );
          } else {       // a < b
               System.out.println(„2. rekursiver Aufruf“);
               return ggT( a, b-a );
          }
     }

     public static void main(String [] argv){
       System.out.println( ggT(27,18));
     }
}
```

- Java-Programme bestehen aus einer oder mehreren *Klassen*
- Eine Klasse steht in der Regel in einer eigenen, gleichnamigen Datei mit der Endung `.java`, z. B. `HelloWorld.java`
- Die Programmausführung beginnt mit der Methode `main` der Klasse, mit der der Java-Interpreter gestartet wird.

2 Grundlegende Sprachkonzepte

2.1 Ausdrücke, Anweisungen, Kommentare

- **Ausdrücke** (engl. *Expressions*) werden *ausgewertet* und haben einen *Typ* (s. 2.4) und einen *Wert*:

  `a == b` ist ein Ausdruck mit einem boolschen Wert

- **Anweisungen** (engl. *Statements*) hingegen werden *ausgeführt* haben *Effekte*

  `return ggT(a-b,b);` ist eine Anweisung, deren Ausführung den Rücksprung aus einer Methode bewirkt.

- Anweisungen werden mit einem Semikolon beendet.

- *Folgen von Anweisungen* (*blocks*) werden durch Einschluß in geschweifte Klammern gebildet und gelten syntaktisch als *eine* Anweisung, d.h. können überall dort verwendet werden, wo sonst genau eine Anweisung erlaubt wäre.

```
...
     else if( a > b ) {
          System.out.println(„1. rekursiver Aufruf“);
          return ggT( a-b, b );
     } else {       // a < b
          System.out.println(„2. rekursiver Aufruf“);
          return ggT( a, b-a );
     }
...
```

- **Kommentare** sind Zeichenfolgen, die das Verständnis des Programmtextes erleichtern sollen, die aber vom Übersetzer ignoriert werden: Kommentare haben somit keine Semantik (im Sinne der Programmausführung), wohl aber Bedeutung!

  `// Kommentar` : die Zeichen vom `//` bis zum Zeilenende gelten als Kommentarttext

  `/* Kommentar */` : die Zeichen zwischen `/*` und `*/` werden ignoriert, auch über Zeilengrenzen hinaus. Solche Kommentare können nicht ineinander verschachtelt werden!

  `/** Kommentar */` : wie oben, können aber von besonderen Dokumentationswerkzeugen interpretiert werden (s. 3.2.4.1)

2.2 Klassen und Objekte

- Klassen sind *statische* Einheiten: sie existieren als Programmtext zur Übersetzungszeit.
- Objekte sind *dynamisch*: sie existieren zur Laufzeit des Programms im Speicher
- Klassen definieren einerseits *Typen* für Objekte und sind andererseits „Schablonen“ zum Erzeugen von Speicherobjekten

Beispiel:

```
public class Fahrzeug{
     public String name = „<unbenannt>;
     public String besitzer;
     public int id;
}
```

- Die Klasse `Fahrzeug` definiert einen *Typ* Fahrzeug mit den drei Attributen `name, besitzer` und `id`.
- Mit der Klasse Fahrzeug kann man *Objekte* (Exemplare, *Instances*) erzeugen:

```
Fahrzeug auto;
auto = new Fahrzeug();
```

1. Die erste Zeile ist die Deklaration einer Variablen vom Typ `Fahrzeug`, d.h. die Variable `auto` ist eine zunächst undefinierte Referenz (Wert: `null`) auf ein `Fahrzeug`-Objekt
2. Die zweite Zeile erzeigt ein Objekt: die Variable `auto` enthält nun eine Referenz auf ein neu erzeugtes Fahrzeug-Objekt

- Die Deklaration der Variablen und Erzeugung eines Objekts kann auch auf einer Zeile erfolgen:

```
Fahrzeug fahrrad = new Fahrzeug();
```

- Objekte haben:
  - einen *Zustand* oder Wert, bestimmt durch den Wert der Variablen der einzelnen Objekte (Instanzvariablen).
  - eine jeweils eigene *Identität*: `fahrrad` und `auto` sind Referenzen auf verschiedene Objekte, obwohl beide Objekte den gleichen (initialen) Zustand haben
  - ein *Verhalten* oder eine Funktionalität, definiert durch die Methoden der Klasse

(Ein besonderes Merkmal von Objektorientierung ist die Integration von Datenstrukturen und Algorithmen im Sprachkonstrukt *Klasse*: Klassen definieren *abstrakte Datentypen*.)

2.2.1 Objektzustand: Instanzvariablen

- Die Klasse `Fahrzeug` vereinbart drei Instanzvariablen mit jeweils einem *Typ*, einem *Namen* und einem *Wert*. Instanzvariablen werden beim Erzeugen des Objekts entweder mit einem angegebenen Wert initialisiert oder mit einem Standardwert:

  `name` ist vom Typ `String` und hat den Wert „<unbenannt>„
  `id` ist vom Typ `int` und hat den Wert 0
  `besitzer` ist vom Typ String und hat den Wert `null`

- Über die Referenz `auto` kann man auf die Instanzvariablen zugreifen:

```
auto.name = „Opel Astra“;
auto.besitzer = „brose“;
System.out.println(„ID: „ + auto.id );
```

- Die Instanzvariablen verschiedener Objekte sind unabhängig, denn jedes Objekt hat einen *eigenen* Zustand, der durch die Werte seiner Instanz-variablen bestimmt ist: `fahrrad.name` ist noch "<unbenannt>", während `auto.name` bereits gesetzt ist:.

2.2.1.1 Klassenvariablen

- Im Unterschied zu Instanzvariablen, die zu einzelnen Objekten gehören, gibt es *Klassenvariablen*, die einer Klasse zugeordnet sind und somit nicht Teil des Zustands eines Objekts, sondern eines globalen Zustands sind.

- Klassenvariablen werden beim Laden der Klasse initialisiert und nicht bei der Erzeugung eines bestimmten Objekts.

- Klassenvariablen werden als `static` markiert:

```
public class Fahrzeug {
     public String name = „<unbenannt>;
     public String besitzer;
     public int id;
     public static int nextID = 0;
}
```

- auf Klassenvariablen wird nicht über eine Objektreferenz, sondern mit dem Klassennamen zugegriffen.

```
auto.id = Fahrzeug.nextID;
Fahrzeug.nextID++;
```

2.2.2 Objekterzeugung: Konstruktoren

- Für die Initialisierung des Zustands bei der Objekterzeugung können lediglich einfache Ausdrücke angegeben werden, aber keine Berechnungsvorschrift mit Parametern. Wenn eine komplexere Berechnung nötig ist, definiert man einen *Konstruktor*:

```
public class Fahrzeug{
   ...;
   public Fahrzeug(){            // Konstruktor
        id = nextID;
        nextID = nextID + 1;     // nextID++
   }
}
```

(In diesem einfachen Fall ist derselbe Effekt auch ohne Konstruktor zu erreichen)

- Bei der Erzeugung eines Fahrzeug-Objekts werden zuerst die Variablen initialisiert, dann wird der angegebene Konstruktor ausgeführt.

- Ist kein Konstruktor definiert, wird ein *impliziter* Konstruktor ohne Argumente angenommen. (sobald ein expliziter Konstruktor definiert ist, fällt der implizite Konstruktor weg!)

- Konstruktoren:

  – tragen den Namen der Klasse

  – können 0 oder mehrere Argumente haben

  – haben keinen Rückgabewert

```
public class Fahrzeug{
   ...;
   public Fahrzeug(String n, String b){
        name = n;
        besitzer = b;
        id = nextID++;
   }
}
```

2.2.3 this

- Das Schlüsselwort `this` bezeichnet immer eine Referenz auf das aktuelle Objekt selbst.

- `this` kann z.B. im Konstruktor verwendet werden, um durch Argumentnamen „verschattete“ Variablennamen zu erreichen:

```
public class Fahrzeug{
   ...;
   public Fahrzeug(String name, String besitzer){
        this.name = name;
```

```
        this.besitzer = besitzer;
        this.id = nextID++;
    }
}
```

2.2.4 Objektverhalten: Methoden

- Die Funktionalität einer Klasse von Objekten (bzw. deren Verhalten) wird von *Methoden* implementiert, z.B. das Setzen oder Auslesen des Objektzustands:

```
public class Fahrzeug{
   ...
   public int geschwindigkeit;

   public void beschleunige(int b){
        geschwindigkeit += b;
   }
   public String toString(){
        return "Fahrzeug " + name + "(" + id + ")";
   }
   ...
}
```

- Der Aufruf einer Methode eines Objekts geschieht genau wie der Zugriff auf eine Instanzvariable mittels des .-Operators:

```
auto.beschleunige(5);
System.out.println( auto.toString() );
System.out.println( auto ); // toString() wird aufgerufen!
```

2.2.4.1 Parameterübergabe

- Bei Aufruf der Methode `beschleunige()` wird nicht ein *Verweis* auf die Variable b, sondern ihr *Wert* übergeben, der als Kopie innerhalb der Methode unter dem Namen b verfügbar ist. Eine Zuweisung an die lokale Kopie b ändert daher nichts an der Variablen `speed`.

```
public class Fahrzeug {
   public void beschleunige(int b){
        geschwindigkeit += b;
        b = 0;
   }
   ...
}
```

```
   int speed = 7;
   auto.beschleunige(speed);
```

- In Java werden *alle* Parameter per *Wertübergabe* (*call-by-value*) an Methoden übergeben, auch Objektreferenzen:

```
public class Person{
   public String name;
```

```
   public Person(String name){
        this.name = name;
   }

   public String toString(){
        return name;
   }
}

public class Fahrzeug {
   ...;
   Person besitzer;
   public void verkaufeAn(Person p){
        besitzer = p;
        p.name = „Otto“;
        p = null;
   }
}
```

- Beim Aufruf von `verkaufeAn()` wird eine *Kopie der Objektreferenz* übergeben, die auf dasselbe Objekt verweist wie die Variable `clara`. Änderungen am Zustand von `clara` sind daher außerhalb der Methode sichtbar, Änderungen an der Referenz (also z.B. die Zuweisungen von `null`) nicht!

```
public class Test {
        public static void main(String [] argv){
             Person paul = new Person(„Paul“);
             Fahrzeug auto = new Fahrzeug(„auto“, paul);
             Person clara = new Person(„Clara“);
             auto.verkaufeAn( clara );
             System.out.println( clara );
        }
   }
```

- Effekt:

```
$ java Test
        Otto
```

2.2.4.2 Überladen (*Overloading*) von Methoden

- Eine Klasse kann den Namen einer Methode überladen, indem mehrere Methoden gleichen Namens, aber mit unterschiedlichen Argumenten definiert werden:

```
public class String {
        public int indexOf(String str)
        {...}
        public int indexOf(String str, int from)
        {...}
   }
```

- Anhand der tatsächlich verwendeten Argumente kann entschieden werden, welche Methode aufgerufen wird.

- Vorteil: man muß sich nicht für jede Variante einer Methode neue Namen ausdenken.

2.2.4.3 Klassenmethoden

- Analog zu Klassenvariablen gibt es *Klassenmethoden*:

```
public class Fahrzeug {
   ...
   public static void setNextID(int i) {
        nextID = i;
   }
}
```

- Klassenmethoden werden mit `static` markiert und werden über den Klassennamen aufgerufen:

```
Fahrzeug.setNextID(17);
```

- Klassenmethoden haben weder Zugriff auf Instanzvariablen, noch können direkt Methoden der eigenen Klasse aufgerufen werden, die nicht selbst Klassenmethoden sind. Innerhalb von Klassenmethoden gibt es kein `this`.

2.2.4.4 Access Modifiers

- Im vorigen Beispiel konnten Objekte anderer Klassen direkt auf Instanzvariablen von Person-Objekten zugreifen und so beliebig den Zustand ändern, was nicht immer erwünscht ist.

- Der Zugriff auf Methoden und Instanzvariablen kann durch die *Modifier* `public, protected, private` gesteuert werden:

  – `public`: Methoden/Variablen sind überall zugänglich

  – `private`: Methoden/Variablen sind nur innerhalb der eigenen Klasse zugänglich

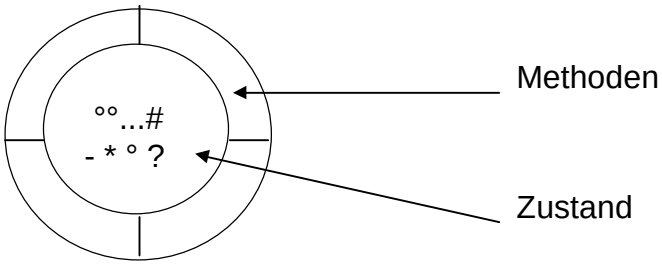
  – `protected`: Methoden/Variablen sind in anderen Klassen desselben Packages (s. 2.7) und in Unterklassen zugänglich

  – kein Modifier: Methoden/Variablen sind nur für Code im selben Package zugänglich.

```
public class Person{
     private String vorname, nachname;
     private int geburtstag;
     private Person mutter, vater;

     public Person(String vorname, String nachname,
                   Person mutter, Person vater,
                   int geb_tag){
          this.vorname = vorname;
          this.nachname = nachname;
          this.mutter = mutter;
```

```
            this.vater = vater;
            this.geburtstag = geb_tag;
      }

      public String vorname(){
            return vorname;
      }
      public int geburtstag(){
            return geburtstag;
      }
      public void adoptiere(Person mutter, Person vater){
            this.mutter = mutter;
            this.vater = vater;
      }
      protected void aendereNachnamen( String name ){
            this.nachname = name;
      }
}
```

- Durch die Einschränkung des Zugriffs wird die Änderung von Instanzvariablen durch Objekte anderer Klassen in kontrollierter Weise erlaubt: Man spricht von einer *Kapselung* des Objektzustands.



(Kapselung schützt so den Objektzustand vor „unsachgemäßer“ Änderung und unterstützt *Datenabstraktion*, indem sie das Verbergen der Repräsentation eines Objekts erlaubt, was die Implementierung des Objekts und seine Schnittstelle entkoppelt.)

2.3 Kontrollstrukturen

Kontrollstrukturen sind Anweisungen, die die Beeinflussung des Kontrollflusses durch das Programm erlauben.

2.3.1 Wiederholungsanweisungen (Schleifen)

Die wiederholte Ausführung anderer Anweisungen.

2.3.1.1 While-Anweisung

While-Anweisungen wiederholen die Ausführung einer Anweisung, solange eine gegebene Bedingung erfüllt ist:

```
while( boolean-expression )
     statement
```

Zunächst wird die Bedingung ausgewertet. Ist sie wahr, wird die Anweisung (bzw. der Block) im Schleifenrumpf ausgeführt.

```
while( x != null ){
     doSomething(x);
     x = getX();
}
```

Während bei der while-Anweisung die Bedingung ausgewertet wird, *bevor* der Schleifenrumpf ausgeführt wird, wird bei der do-while-Anweisung zunächst der Schleifenrumpf ausgeführt und *anschließend* die Bedingung überprüft:

```
do {
     doSomething(x);
     x = getX();
} while ( x != null )
```

2.3.1.2 For-Anweisung

For-Anweisungen erlauben neben der Wiederholung einer Anweisung das automatische Ausführen einer „Inkrementierungs-Anweisung“ bei jedem Durchlauf:

```
for( init-expr; boolean-expression; incr-expression  )
     statement
```

Die `init-expression` wird einmal zu Beginn ausgeführt. In den Schleifenrumpf gelangt die Ausführung nur, wenn die Bedingung erfüllt ist. Nach jedem Durchlauf durch den Rumpf wird die Anweisung in der `incr-expression` einmal ausgeführt und die Bedingung erneut geprüft:

```
for( int i = 0; i < 4711; i++ )
     sum += i;
```

Jede for-Anweisung läßt sich auch als while-Anweisung formulieren:

```
   init-expr;
   while( boolan-expression ) {
      statement
      incr-expression;
   }
```

Also z.B.:

```
int i = 0;
while( i < 4711 ){
     sum += i;
     i++;
}
```

Alle Expressions der for-Anweisung sind optional. Für eine `fehlende boolean-expression` wird `true` angenommen. Auf diese Weise lassen sich auch Endlosschleifen formulieren:

```
for(;;)        // „forever“
     statement
```

Endlosschleifen werden z.B. verwendet, wenn es nicht nur ein, sondern mehrere Abbruchkriterien gibt, bei deren Eintreten die Schleife dann durch eine explizite *Sprunganweisung* (s.u.) verlassen wird.

2.3.2 bedingte Anweisungen (Verzweigung)

2.3.2.1 If-Anweisung

Alternative Auswahl einer Anweisung je nach dem Wert eines *boolschen* Ausdrucks:

```
if( x > y )
     return x;
else
     return y;
```

2.3.2.2 Switch-Anweisung

Die switch-Anweisung erlaubt die Verzweigung in Abhängigkeit vom Wert eines *ganzzahligen* Ausdrucks, der wird mit einer Reihe von Konstanten verglichen wird:

```
char c = getACharacterFromSomewhere();
String result = null;
switch( c ){
     case ‘a’ :
          result = „Es war ein A.“;
          break;
     case ‘q’ :
          result = „Es war ein q“;
     default  :
          result = „Nur a oder q war erlaubt.“;
}
```

Achtung: die Kontrolle kehrt nicht automatisch aus den `case`-Zweigen zurück! Ohne Sprunganweisung wird einfach die nächste Anweisung ausgeführt, also in obigem Beispiel die Anweisung, die zum nächsten Zweig gehört! Der Sinn dieser Eigenschaft der switch-Anweisung liegt darin, daß sich auf diese Weise mehrere Zweige „zusammenfassen“ lassen:

```
char c = getACharacterFromSomewhere();
String result = null;
switch( c ){
     case ‘a’ :
     case ‘b’ :
     case ‘c’ :
     case ‘d’ :
          result = „Es war a-d.“;
          break;
     case ‘l’ :
     case ‘m’ :
     case ‘n’ :
     case ‘o’ :
          result = „Es war l-o.“;
          break;
     default  :
          result = „Kein gültiges Zeichen!“;
}
```

2.3.3 Sprunganweisungen

- `break`: verläßt den umschließenden Block, z.B. in switch-Anweisungen, aber auch in jedem anderen Block:

```
for( int i = 0; i < schwelle; i++ )
   if( found() )
        break;
```

- `continue`: springt zum Ende des aktuellen Schleifenrumpfs (for, while, do-while) und wertet die Schleifenbedingung erneut aus:

```
while( notDone()){
   int i = getValue();
   if( value == 4711 ) // 4711 wird übersprungen
        continue;
   doSomethingLongAndComplicated(i);
   process(i);
}
```

- `return`: beendet die Ausführung einer Methode und liefert ggf. einen Rückgabewert. Der Rücksprung aus einer Methode verläßt eventuell weiter umgebende Blöcke (Schleifen).

2.4 Typen in Java

bisher gesehen: `String, int, Person, Fahrzeug`

2.4.1 Wozu Typen?

- Typen schränken ein, welche Werte eine bestimmte Variable annehmen kann:

```
int eineZahl;
eineZahl = 'a'; // Compiler meldet Fehler!
```

- Hardware kennt zunächst nur ungetypte Speicherworte:
  00110100010101110010101000 1...
  -> jeder Wert muß so codiert werden

- „*Man soll Äpfel nicht mit Birnen vergleichen*“: man sieht einem Speicherbereich nicht an, ob er einen Apfel oder eine Birne darstellt (bzw. eine ganze oder eine reelle Zahl, einen Maschinenbefehl, eine Speicheradresse oder ein Zeichen...

- Programmieren mit ungetypten Werten ist möglich, hat aber folgende Nachteile:
  - Für jeden neue Art von Werten muß man sich eine binäre Repräsentation überlegen, die man sowohl beim Erzeugen neuer Werte wie beim Umgang damit verwenden muß.
  - Der Übersetzer kann nicht auf bestimmte Programmierfehler hinweisen, etwa unsinnige Operationen wie: `birne = 1 * true`;

- Hochsprachen haben daher *Typsysteme*:
  - Typen fungieren als Schablonen für die Erzeugung von Speicherobjekten (Werten) -> Arbeitserleichterung
  - Typsysteme führen Beschränkungen ein, die Korrektheit herbeiführen helfen sollen („Man soll Äpfel und Birnen ...“). Der Übersetzer/Interpretierer überprüft die Einhaltung solcher Regeln.

- Java: Typsicherheit (=die Einhaltung aller durch das Typsystem definierten Regeln) wird sowohl (statisch) vom Übersetzer wie (dynamisch) vom Interpretierer überprüft.

- Java ist *streng* typisiert, d.h. *jeder* Ausdruck hat einen wohldefinierten Typ

- Objekttypen werden durch Klassen definiert, Variablen solcher Typen enthalten stets *Referenzen:*

```
Fahrzeug auto, fahrrad;
auto = new Fahrzeug(„auto“, new Person(„paul“));
fahrrad = auto;             // keine Kopie, sondern Alias!
fahrrad.name = „fahrrad“;
System.out.println(auto.name); // „fahrrad“
```

2.4.2 Basistypen und Operatoren:

Java-Basistypen:

- `byte, short, int, long:`

  ganze Zahlen mit Vorzeichen (8,16,32 und 64 Bit)

- `boolean`

  Wahrheitswert `true` oder `false` (1 Bit)

- `char`

  Ein Zeichen (16 Bit, Unicode-Darstellung)

- `float, double`

  Fließkommazahlen nach IEEE-Standard (32 und 64 Bit)

2.4.2.1 Arithmetische Operatoren:

Binär:

`+,-,*,/, %`

Unäre Inkrement/Dekrement-Operatoren a la C/C++ kombinieren das Erhöhen/Verringern des Werts einer Variablen mit der Auswertung:

`++, --`

Bsp.:

```
int a = 5;
int b = 4;
int c = a++;   // c ist 5, a hat den Wert 6
c = ++a;       // c ist 7, a ist 7 („Präinkrement„)
```

2.4.2.2 Vergleichsoperatoren:

`>,<, <=, >=, ==, !=`

Achtung: Vergleich mit == bei Objekttypen vergleicht die *Referenzen*, nicht den Zustand!

```
int a = 5, b = 5;              // a == b
Person p1 = new Person(„otto„);
Person p2 = new Person(„otto„);// p1 != p2
p1 = p2;                       // p1 == p2
```

Um den Wert zweier Objekte einer Klasse zu vergleichen, verwendet man die Methode `equals()`:

```
...
if( p1.equals(p2))
     doSomething();
```

2.4.2.3 Logische Operatoren

arbeiten mit boolschen Argumenten

`&&, ||, !` : UND, ODER, NICHT

`&, |` : *striktes* UND und ODER

Beispiel:

```
if( o1.isReady() && o2.isReady())
     return true
else
     return false
```

Nicht strikt: ist `o1.isReady() false`, so wird `o2.isReady`() nicht mehr aufgerufen, denn der gesamte Ausdruck kann ohnehin nicht mehr `true` werden. (ODER analog)

```
if( o1.isReady() & o2.isReady())
     return true
else
     return false
```

Strikt: `o2.isReady()` wird unabhängig vom Resultat von `o1.isReady()` aufgerufen.

Achtung: Die strikte oder nicht-strikte Auswertung boolscher Ausdrücke kann wegen den Nebenwirkungen der Ausführung von Methoden unterschiedliche Programmsemantik zur Folge haben! Sollte `isReady`() beispielsweise eine Bildschirmausgabe oder eine Zustandsänderung bewirken, so fehlt dieser Effekt unter Umständen bei der nicht-strikten Auswertung.

2.4.2.4 Zuweisungsoperatoren:

`=` einfache Zuweisung eines Werts an eine Variable: a = 5;

`+=, -=, *=, /=`, ... Kurzschreibweisen:

`a += 5;` ist kurz für: `a = a + 5;`

2.4.2.5 Bitoperatoren:

`&, |, ^` bitweises UND, ODER, XOR

`>>, <<` rechts- und linksshift

`~` bitweise Komplementbildung

2.4.3 Arrays

- sind geordnete, homogene Mengen von Werten desselben Typs:

```
// Deklaration
int [] zahlen;
Fahrzeug [] fuhrpark;

//Erzeugung
zahlen = new int[23];
fuhrpark = new Fahrzeug[7];

// Elementzugriff
zahlen[0] = 7;
fuhrpark[0] = new Fahrzeug(...);
System.out.println(„Der Fuhrpark besteht aus “+
            fuhrpark.length + „ Fahrzeugen.“);
```

- Arraytypen sind Referenztypen, d.h. eine Arrayvariable enthält eine *Referenz* auf das Array.

- Die Arraygröße wird bei der Erzeugung festgelegt und ist über das Attribut `length` verfügbar. Sie ist nicht dynamisch änderbar. Das erste Element eines Arrays hat stets den Index 0, das letzte den Index `array.length-1`. Das Lesen über Feldgrenzen hinweg führt zur Laufzeit zu einer Ausnahme (s. 2.6), einer `IndexOutOfBoundsException`.

- Arrays können auch so initialisiert werden:

  ```
  String [] meineFreunde = {„peter“,“paul“,“mary“};
  ```

- Typischer Durchlauf durch ein Array:

```
int[] zahlen = {1,7,5,12,92,67,33};
int min = zahlen[0];
for( int i = 1; i < zahlen.length; i++){
     if( zahlen[i] < min ){
          min = zahlen[i];
     }
}
```

- „Mehrdimensionale“ Arrays sind Arrays, deren Elemente selbst Arrays:

```
int [][] matrix = new int[7][5];
for( int i = 0; i < matrix.length; i++)
     for( int j = 0; j < matrix[i].length; j++ )
          matrix[i][j] = j;
```

2.4.3.1 Arrays sortieren: Quicksort

Ein sehr effizienter Algorithmus zum Sortieren von Feldern ist „QuickSort“ (Hoare). Die Idee: Man unterteilt ein gegebenes, unsortiertes Feld in einen Abschnitt links von einem mittleren Element und einen Abschnitt rechts davon. Nun

werden solange Elemente des linken und rechten Abschnitts vertauscht, bis alle Elemente der linken Hälfte kleiner als das mittlere Element sind und alle Elemente der rechten Hälfte größer. Nun wird das Verfahren für die linke und rechte Feldhälfte rekursiv wiederholt, bis das gesamte Feld sortiert ist.

Beispiel:

7	2	3	4	8	6	1

1	2	3	4	8	6	7

usw.

Quicksort in Java:

```
void quickSort( int l, int r, int [] unsorted){
     int i = l;
     int j = r;
     int x = unsorted[ (l+r)/2 ];
     while ( i <= j ){
          while( unsorted[i] < x )
               i++;
          while( x < unsorted[j] )
               j--;
          if( i <= j ){
               int tmp = unsorted[i];
               unsorted[i] = unsorted[j];
               unsorted[j] = tmp;
               i++;
               j--;
          }
     }
     if( l < j )
          quickSort( l, j, unsorted );
     if( i < r )
          quickSort( i, r, unsorted );
}
```

2.4.4 Strings (Zeichenketten)

Strings sind in Java kein Basistyp, sondern eine Bibliotheksklasse (`java.lang.String`)

```
java.lang.String s = „hallo„;
```

Diese Initialisierung ist äquivalent zu:

```
java.lang.String t = new java.lang.String(„hallo„);
```

Das Konkatenieren (Aneinanderhängen) von Strings mit dem +-Operator funktioniert auch, wenn ein Argument kein String ist. In solchen Fällen werden Argumente in Strings konvertiert (Methode `toString()` bei nicht-Basistypen)

```
int i = 4711;
String str = „Der Wert von i ist:“ + i;
```

Die Länge eines Strings (Anzahl der Zeichen) wird durch die Methode `length()` erfragt, ein bestimmtes Zeichen kann über seinen Index mit `charAt()` erfragt werden:

```
for( int i = 0; i < str.length(); i++)
   System.out.print( str.charAt(i) );
```

2.4.4.1 Strings vergleichen

Um Strings auf Gleichheit zu testen, verwendet man `equals()` bzw. `equalsIgnoreCase()`:

```
String str = „HelloWorld„.
String s2 = „helloworld„;
if( str.equals( s2 ))
     System.out.println( „equals„ );
else if( str.equalsIgnorCase( s2 ))
     System.out.println( „equalsIgnoreCase„ );
else
     System.out.println( „equalsIgnoreCase„ );
```

Um Strings dem Wert nach sortieren zu können, vergleicht man mittels der Methode `compareTo()`, einen Wert liefert, der kleiner, gleich oder größer als 0 ist, je nachdem, ob der String, dessen `compareTo()`-Methode aufgerufen wurde, kleiner, gleich oder größer als der andere String ist.

Beispiel: Suche in einem Feld.

Bei der *linearen Suche* nach einem Element in einem unsortierten Feld wird das Feld von vorne bis hinten durchlaufen und jedes Element mit dem gesuchten verglichen. Bei Gleichheit ist das Element gefunden, wenn das Feldende erreicht wurde, ist es nicht gefunden.

```
private String [] table;
public int positionLinear(String key){
     for( int i = 0; i < table.length; i++)
          if( table[i].equals(key) )
               return i;
}
```

Aufwand: im Mittel sind bei einer Feldlänge von n Elementen n/2 Vergleiche notwendig -> der Aufwand steigt linear mit der Feldlänge

Binäre Suche: kann ein sortiertes Feld vorausgesetzt werden, so wird das gesuchte Element mit einem Element aus der Feldmitte verglichen. Bei Gleichheit ist das Resultat gefunden. Falls das gesuchte Element kleiner als das verglichene ist, wird mit der Suche in der unteren Hälfte des Feldes fort-

gefahren und die obere nicht mehr betrachtet. Ist es größer, wird in der oberen Hälfte weitergesucht. (vgl. suchen im Telefonbuch)

```
public int positionBinary( String key ){
     int lo = 0;
     int hi = table.length – 1;
     while( lo <= hi ){
          int mid = lo + ( hi - lo ) / 2;
          int cmp = key.compareTo(table[mid] );
          if( cmp == 0 )      // gefunden
               return mid;
          else if ( cmp < 0 ) // im unteren Teil weiter
               hi = mid - 1;
          else
               lo = mid + 1;  // im oberen Teil weiter
     }
     // nicht gefunden
     return -1;
}
```

Aufwand: die Anzahl der Vergleiche wächst bei diesem Verfahren lediglich logarithmisch mit der Größe des Feldes!

2.4.4.2 Strings vs. StringBuffer

Strings in Java sind konstant, d.h. unveränderlich. Die folgenden beiden Zuweisungen, die keine bestehenden Strings manipulieren sondern jeweils *neue* String-Objekte erzeugen, sind daher äquivalent:

```
s = s + „ Welt!„;
s = new String( s + „ Welt!„);
```

Um effizient mit veränderlichen Strings arbeiten zu können, gibt es eine Klasse `StringBuffer`. Mit Operationen wie `append(), insert()` und `setCharAt()` können `StringBuffer`-Objekte manipuliert und geändert werden.

Beispiel:

```
StringBuffer sb = new StringBuffer();
String str = getInputString();
while( str != null ){
     sb.append( str );
     str = getInputString();
}
String result = sb.toString();
```

2.5 Objektorientierung

- Def. (P. Wegner):
  Objektorientierung = Klassen + Vererbung + Polymorphie
- Klassen schon bekannt, erlauben Kapselung.

2.5.1 Vererbung

ist eine Erweiterungsbeziehung zwischen Klassen zur Wiederverwendung von Code:

```
public class Person {
     String vorname, nachname;
     int geburtstag;
     ...
     public String nachname(){
          return nachname;
     }
}


public class Professor extends Person {
     Fachbereich fachbereich;
     int besoldungsgruppe;
     public void emeritiere(){
          ...
     }
}
```

- `Professor` ist eine Unterklasse („Spezialisierung“, „Erweiterung“) von `Person`. Sie „erbt“ die Variablen und Methoden der Klasse `Person` und erweitert ihre Oberklasse um die Variablen `fachbereich` und `besoldungsgruppe` sowie die Methode `emeritiere`()
- In Java gibt es nur *Einfachvererbung*, d.h. eine Klasse kann nur *eine* Oberklasse haben.
- Alle Klassen erben direkt oder indirekt von der Klasse `Object`, die einige Merkmale definiert, die für alle Objekte gelten. (d.h. auch, daß jede Klasse bis auf `Object` *genau* eine Oberklasse hat.)

2.5.2 Polymorphie (Vielgestaltigkeit)

- Objekte einer Unterklasse sind auch legale Exemplare der Oberklasse: ein Professor ist eine Person, d.h. einem `Person`-Objekt kann auch ein `Professor`-Objekt zugewiesen. Professoren können überall verwendet werden, wo Personen erwartet werden:

```
Professor prof = fu.getDekan(„chemie“, 1997);
Person p = prof;
System.out.println(p.nachname() + „geb. “ +
                   p.geburtstag );
```

- Hinter einem `Person`-Objekt können also Objekte beliebiger Unterklassen von `Person` stecken.

- hinter einem Objekt vom Typ `Object` können Objekte beliebigen Typs stekken! -> Generische Methoden (Methoden, die für alle Objekttypen funktionieren) haben daher Argument- oder Resultattypen `Object`.

  Beispiel: Bibliotheksklasse `java.util.Vector`

  - Arrays haben nach ihrer Erzeugung feste Feldgrenzen und können nicht dynamisch wachsen. Die Bibliotheksklasse `java.util.Vector` ist eine dynamisch erweiterbare, generische Folge.

  - Die Methode `addElement()` erwartet Objekte vom Typ `Object`,

```
Vector v = new Vector();
while( (String str = getValue()) != null )
     v.addElement( str );
```

2.5.2.1 Dynamische Typumwandlung (Cast)

- durch die Zuweisung einer Objektreferenz an eine Variable von einem allgemeineren Typ geht zunächst Typinformation verloren:

  Eine Variable vom Typ Person, hinter der ein `Professor`-Objekt steckt, kann z.B. nicht mehr einer Methode als Argument übergeben werden, die ein `Professor`-Objekt erwartet, obwohl es tatsächlich ein solches ist.

```
public void kooperiere(Professor prof);
...
Professor professor = new Professor(...);
professor.kooperiere( p );// statischer Typfehler!
```

  Daher gibt es die Möglichkeit der dynamischen Typumwandlung durch sogenannte „casts“. Im folgenden Beispiel wird versucht, die Variable p nach `Professor` zu „casten“, was nur gelingt, wenn das referenzierte Objekt tatsächlich von diesem Typ oder einer Unterklasse davon ist.

```
public void kooperiere(Professor prof);
...
Professor professor = new Professor(...);
professor.kooperiere( (Professor)p );
// dynamischer Typfehler, falls p kein Professor!
```

```
//Vector-Beispiel:
String [] values = new String[v.size()];
for( int i = 0; i < values.length, i++ )
      values[i] = (String)v.elementAt(i);
```

- wie kann man vorher feststellen, ob der Cast gelingt oder nicht? -> der Operator `instanceof` prüft, ob ein Objekt ein Exemplar der angegeben Klasse ist.

```
if( p instanceof Professor )
     professor.kooperiere( (Professor)p); // geht immer gut
else
 ... // tue was anderes
```

2.5.2.2 Wrapperklassen

- Referenztypen (Objekttypen) können nicht auf Basistypen gecastet werden (und umgekehrt). Wenn man Basistypen an einer Stelle verwenden will, wo nur Objekttypen erlaubt sind, kann man den Wert eines Basistyps in ein passendes „Wrapper"-Objekt einpacken.

- Für jeden Basistyp gibt es eine entsprechende Wrapper-Klasse, für `int` z.B. `java.lang.Integer`:

```
Vector v = new Vector();
while( (int i = getValue()) >= 0 )
     v.addElement( new Integer(i));
```

2.5.3 Konstruktoren in Unterklassen

- Konstruktoren werden nicht vererbt, d.h. Unterklassen müssen eigene Konstruktoren angeben und ggf. die Oberklassenkonstruktoren benutzen:

```
public Professor(String vorname, String nachname,
                 Person vater,Person mutter, int geburtstag,
                 Fachbereich fachbereich,
                 int besoldungsgruppe)
{
     super(vorname,nachname,vater,mutter,geburtstag);
     this.fachbereich = fachbereich;
     this.besoldungsgruppe = besoldungsgruppe;
}
```

- das Schlüsselwort `super` ist eine Referenz auf das Objekt *selbst als Exemplar seiner Oberklasse*

- der implizite default-Konstruktor (s. 2.2.2) enthält den Aufruf von `super()`

2.5.4 Überschreiben/Overriding von Methoden

- Unterklassen können Methoden der Oberklasse *überschreiben* (*overriding*), d.h. der Name der Methode der Unterklasse „verschattet" den Namen der Methode der Oberklasse – die Methode ist aber noch vorhanden.

- die überschreibende Methode muß dieselbe Signatur wie die Methode der Oberklasse haben:

```
public class Person {
     ...
     public String nachname(){
          return nachname;
     }
```

```
}

public class Professor extends Person {
     ...
     public String nachname(){
          return „Professor „ + nachname;
     }
}
```

- Der *aktuelle Typ* des Objekts bestimmt, welche Methode aufgerufen wird:

```
Person p = new Person(„Helmut“,“Kohl“,... );
Person prof = getDekan(...);
System.out.println( p.nachname );
System.out.println( prof.nachname );
```

Effekt:

```
Kohl
Professor Müller
```

- in der überschreibenden Methode kann man auf die überschriebene Methode zugreifen:

```
public String nachname(){
     return „Professor „ + super.nachname();
}
```

- überschreibende Methoden können nur *mehr* Zugriffsrechte haben, nicht weniger: `private -> protected -> public`
- Eine als `final` markierte Methode kann nicht in Unterklassen überschrieben werden:

```
public final String nachname()
```

- Von einer als final markierten Klasse können keinen Unterklassen gebildet werden:

```
public final class String { ... }
```

- Auch Instanzvariablen können „verschattet“ werden:

```
public class Person {
     ...
     public String testString = „<person>“;
}
```

```
public class Professor extends Person {
     ...
     public String testString = „<prof>“;
}
```

```
Professor prof=new Professor("Klaus","Mueller",..);
     Person p = prof;
```

```
     System.out.println("Prof: " + prof.testString );
     System.out.println("Person: " + p.testString );
```

- Achtung: bei Instanzvariablen entscheidet der statische *Typ der Referenz*, nicht der aktuelle Objekttyp!

  ```
  Prof: <prof>
  Person: <person>
  ```

- `testString` in Professor verdeckt `testString` in `Person`

2.5.5 Abstrakte Klassen

- abstrakte Klassen sind Klassen, deren Implementierung teilweise noch offengelassen ist: bestimmte Methoden sind zwar deklariert, aber noch nicht implementiert.
- Abstrakte Klassen können nicht instanziiert werden (= es können keine Objekte erzeugt werden).
- Die fehlende Implementierung wird in Unterklassen vervollständigt.

```
abstract class Benchmark {
     abstract void benchmark();

     public long repeat(int count){
          long start = System.currentMillisec();
          for(int i = 0; i < count;i++)
               benchmark();
          return (System.currentMillisec() - start);
     }
}

class MethodBenchmark extends Benchmark {

     void benchmark()
     {}

     public static void main(String [] args){
          int count = Integer.parseInt(args[0]);
          long time = new MethodBenchmark().repeat(count);
          System.out.println(count + „ methods in „ +
                         time + „ milliseconds“);
     }
}
```

- `MethodBenchmark` mißt die Zeit eines einfachen (leeren) Aufrufs.
- Das Array `args` enthält die Argumente, die beim Start des Programms auf der Kommandozeile übergeben wurden
- `Integer` ist eine Bibliotheksklasse, `parseInt()`eine statische Methode, die einen `String` in ein `int` umwandelt.

2.5.6 Interfaces (Schnittstellen)

- Interfaces sind vollständig abstrakte Klassen, d.h. Typdefinitionen ohne Implementierung (-> „Datenabstraktion“)

```
interface Menge {
     public void einfuegen(Object o);
     public void herausnehmen(Object o);
     public boolean istEnthalten(Object o);
}
```

- Ein Benutzer eines Typs Menge kennt nicht die *Repräsentation* dieses Typs, nur die Operationen, mit denen auf Mengen zugegriffen werden kann. Vorteil: Es sind verschiedene Implementierungen desselben Typs möglich, und die Implementierung kann geändert werden, ohne daß ein Benutzer es merkt!

- Alle Methoden eines Interfaces sind implizit `public` und `abstract`, niemals jedoch `static`. Alle (Attribute) Variablen sind implizit `public`, `static` und `final` und erlauben die Definition von Konstanten

- Interfaces erlauben *Mehrfachvererbung*: eine Klasse kann beliebig viele Interfaces implementieren bzw. ein Interface kann beliebig viele andere Interfaces erweitern – ein Typ kann also mehrere Supertypen haben!

- Beispiel: Eigenschaftstypen mit verschiedenen Merkmalen:

```
interface Sortable {
     boolean greaterThan(Sortable other);
}

interface Cloneable {
     Object clone();
}

interface C extends Sortable, Cloneable {...}

class D implements Sortable, Cloneable {...}
```

- In Java ist zwischen *Implementierungsvererbung* (nur einfach, zwischen Klassen) und *Interfacevererbung* (auch mehrfach, zwischen Interfaces bzw. Interfaces und Klassen) zu unterscheiden!

- Interfaces sind sowohl ein *Abstraktionsmittel* (zum Verbergen von Details) wie ein *Strukturierungsmittel* zur Organisation von Klassenhierarchien!

2.6 Ausnahmebehandlung

- *Ausnahmen* (engl. *Exceptions*) sind unerwartet auftretende Laufzeitfehler, z.B. Division durch null, Lesen über das Ende einer Datei hinaus, Zusammenbrechen einer Netzwerkverbindung, Lesen über Arraygrenzen hinweg, etc.

- Beim Auftreten einer Ausnahme wird die Ausführung des Programms unterbrochen und ein Ausnahmeobjekt „geworfen“ (*throw*)

- Es wird nach einer passenden Ausnahmebehandlung gesucht, die das Ausnahmeobjekt „auffängt“ (*catch*) -- wird keine Ausnahmebehandlung gefunden, wird das Programm abgebrochen.

Beispiel:

```
public class Person {
     ...
     protected void aendereNachnamen( String name )
          throws InvalidNameException
     {
          if( name == null )
               throw new InvalidNameException(name);
          this.nachname = name;
     }
}
...
try {
     paul.aendereNachnamen ( null );
} catch ( InvalidNameException i){
     i.printStackTrace();
     // gib Fehler aus und fahre fort
}
```

- Ausnahmen können in der Methodendeklaration aufgelistet werden, so daß der Übersetzer überprüfen kann, ob der aufrufende Code

  - eine entsprechende Ausnahmebehandlung vorsieht, also eine passende `catch`-Anweisung hat, oder

  - selbst eine Methode mit entsprechender Ausnahmevereinbarung ist, also ebenfalls eine `throws`-Klausel hat

- Exceptions sind Objekte einer Exceptionklasse und erben entweder von `java.lang.Exception` oder `java.lang.RuntimeException.`

- Exceptions, die Unterklassen von `RuntimeException` sind, sind sogenannte „unchecked exceptions“ – sie müssen nicht in einer `throws`-Klausel deklariert werden, und der Übersetzer verlangt keine Ausnahmebehandlungen.

2.7 Packages

- Packages sind ein Gruppierungskonstrukt für Interfaces, Klassen und Unter-Packages in Bezug auf Namen und Zugriffsrechte.
- Jede Klasse ist in genau einem Package: In einem *Package-statement* kann ein Package genannt werden, zu dem die Klasse gehören soll. Wird kein Package genannt, so gehört die Klasse ins globale, unbenannte Package.

```
package kurs;
public class X ...
```

2.7.1 Klassennamen und Packages

- Der vollständig qualifizierte Name einer Klasse wird aus dem Klassennamen und allen umschließenden Package-Namen gebildet, z.B. `java.awt.event.MouseEvent`
- Innerhalb eines Packages müssen Klassennamen eindeutig sein.
- Klassen in anderen Packages müssen mit ihrem vollständig qualifizierten Namen angesprochen werden, außer wenn:
  - die Klasse im Package `java.lang` ist (z.B. `java.lang.String`)
  - eine entsprechende Import-Anweisung vorhanden ist

```
package kurs.beispiele;
import kurs.util.*;
import java.util.Hashtable;

public class Person ...
          java.util.Vector v = new java.util.Vector();
          Hashtable h = new Hashtable();
...
```

2.7.2 Packages und Zugriffsregeln

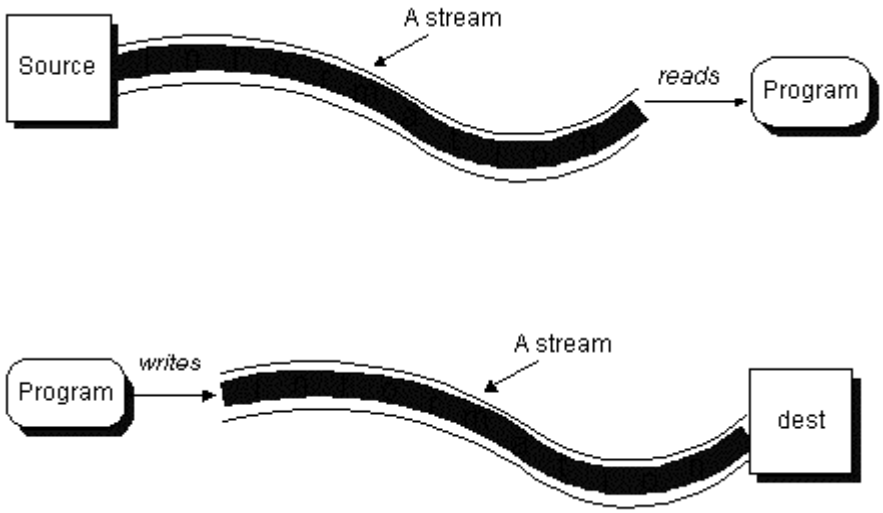
- Durch geeignete *access modifiers* (s. 2.2.4.4) lassen sich *Felder und Methoden* so markieren, daß sie nur von Code innerhalb desselben Packages aus verwendbar sind.
- Wenn *Klassen* als `public` markiert sind, sind sie außerhalb des eigenen Packages verwendbar. Ist eine Klasse nicht `public`, kann sie nur im eigenen Package verwendet werden.

2.8 Ein-/Ausgabe

Ein-/Ausgabe in Java geschieht mit Strömen (*Streams*), die den Programmierer von den Betriebssystemdetails (etwa des Dateisystems) abschirmen. Diese abstraktere Ein-/Ausgabeschnittstelle ist auch auf unterschiedlichen Plattformen (fast) gleich.

2.8.1 Streams

- Streams sind Sequenzen von Werten, ähnlich Listen, auf die sequentiell lesend oder schreibend zugegriffen wird.



- Java stellt eine Auswahl an Stream-Klassen im Package `java.io` zur Verfügung:
  - *Eingabeströme* sind Unterklassen der abstrakten Klasse `InputStream`, von ihnen wird gelesen
  - *Ausgabeströme* sind Unterklassen der abstrakten Klasse `OutputStream`, in sie wird geschrieben.
- Das folgende Beispiel kopiert Zeichen von der Standardeingabe in die Standardausgabe:

```
import java.io.*;
public class Echo {
  public static void main(String [] argv ){
       InputStream in = System.in;
       OutputStream out = System.out;
       try {
            int ch = 0;
            while( (ch = in.read()) != -1 ) {
                 out.write( ch );
            }
```

```
        } catch ( IOException e ) {
            e.printStackTrace();
        }
    }
}
```

- Die Streams `System.in` und `System.out` sind die Standardeingabe (Tastatur) und die Standardausgabe (Bildschirm).
- Die Operation `read()` liefert eine Ganzzahl zwischen 0 und 255 (ein Byte) und `-1` bei Stromende. Der Aufruf von `read()` blockiert, bis ein Datum vorliegt. Die Operation `write()` schreibt eine Ganzzahl als Byte (nur die niederwertigen 8 Bit werden übertragen!) und blockiert, bis das Zeichen geschrieben ist. Sowohl `read()` wie `write()` werfen `IOExceptions` bei Ein-/Ausgabefehlern (z.B. Verbindungszusammenbruch etc.):

  ```
  public abstract int read() throws IOException
  ```

- Ein-/Ausgabe mit Dateien geschieht mit speziellen Unterklassen von `Input/OutputStream`, z.B. `FileInputStream`:

```
import java.io.*;
public class Echo {
   public static void main(String [] argv ){
        try {
            InputStream in;
            if( argv.length == 0 )
                in = System.in;
            else
                in = new FileInputStream( argv[0] );
            OutputStream out = System.out;
            int ch = 0;
            while( (ch = in.read()) != -1 ){
                out.write( ch );
            }
        } catch ( IOException e ) {
            e.printStackTrace();
        }
   }
}
```

- Ströme lassen sich auch kombinieren, indem ein neuer Strom als „Filter“ vor ein anderes Strom-Objekt gelegt wird. Die Klasse `DataOutputStream` stellt beispielsweise komfortablere Ausgabemethoden bereit als `OutputStream.` Damit lassen sich die Werte aller Basistypen (nicht nur Bytes) direkt ausgeben.
- Beispiel: Aus einem gegebenen `OutputStream` läßt sich leicht ein `DataOutputStream` machen:

```
import java.io.*;
public class Echo {
   public static void main(String [] argv ){
        try{
            InputStream in;
```

```
            if( argv.length == 0 )
                 in = System.in;
            else
                 in = new FileInputStream( argv[0] );

            DataOutputStream out;
            if( argv.length < 2 )
                 out = new DataOutputStream(System.out);
            else
                 out = new DataOutputStream(
                           new FileOutputStream( argv[1] ));
            out.writeChars("-- begin of output file --\n");
            int ch = 0;
            while( (ch = in.read()) != -1 ){
                 out.write( ch );
            }
            out.writeChars("-- end of output file --\n");
            out.close(); // schliesst den DataOutputStream
        } catch ( IOException e ) {
            e.printStackTrace();
        }
    }
}
```

2.8.1.1 Objektserialisierung

- Auch Objekte können über Streams ein- und ausgegeben („serialisiert“) werden: Die Klassen `ObjectInputStream` und `ObjectOutputStream` bieten die Methoden `readObject()` und `writeObject()`.

- Es können nur Objekte solcher Klassen serialisiert werden, die entweder das Interface `Serializable` oder `Externalizable` implementieren: `Serializable` ist ein leeres Interface, das nur zur Markierung dient. `Externalizable` definiert zwei Methoden, die Unterklassen implementieren können, um selbst Einfluß auf den Serialisierungsvorgang zu nehmen.

- Referenziert ein zu serialisierendes Objekt weitere Objekte, so werden auch diese serialisiert, d.h. es wird u.U. ein kompletter Graph von Objekten übertragen („deep copy“).

- Instanzvariablen, die keine Bedeutung für den Zustand haben (z.B. weil sie stets neu berechnet werden), können als `transient` markiert werden, so daß sie bei der Serialisierung übergangen werden.

3 Fortgeschrittene Konzepte

3.1 Inner Classes

- In Java können Klassen- oder Interfacedefinitionen zur besseren Strukturierung auch verschachtelt werden. (Mehr Beispiele gibt es im Zusammenhang mit GUI-Programmierung.)

3.1.1 Nested classes

```
class A {
   static int i = 4711;
   static class B implements V {
        int my_i = i;
        ...
   }
}
```

- „nested classes“ erlauben eine weitergehende Strukturierung des Namensraums von Klassen. Nested classes werden als `static` markiert.
- sie werden genauso verwendet wie normale Klassen:
  `V v = new A.B();`
- eine „nested class“ hat Zugriff auf die Klassenvariablen ihrer umgebenden Klasse.

3.1.2 Member classes

```
public class Outer {
   String str = "nesty";
   public I getAnI(){
        return new Inner();
   }

   public class Inner implements I{
        public Inner(){
             str = „inner“;
        }
   }
}
```

- „member classes“ erlauben den Zugriff auf *Instanz*variablen eines Objekts der äußeren Klasse
- Objekte von *member classes* sind immer implizit mit einem Objekt der äußeren Klasse verbunden:

```
Outer o = new Outer();
Outer.Inner i = o.new Inner();  // !!!!
```

- Nützlich in Fällen, in denen eine Adapterklasse benötigt wird, deren Objekte „Hilfsdienste“ für die Basisklasse leisten, z.B. bei der Ereignisbehandlung.

3.1.3 Local classes

```
public class C {
   ...
   public void doSomething(){
        int i = 0;
        class X implements Y { // „local class“
             public X(){...}
             public void perform(){
                  ...
             }
        }
        Y y1 = new X();
        y1.perform();
   }
}
```

- *local classes* können an beliebiger Stelle im Code definiert werden
- *Sie* werden verwendet, wenn eine Klasse nur im lokalen Kontext gebraucht wird, so daß die Definition einer global sichtbaren Klasse überflüssig ist.

3.1.4 Anonymous Classes

```
public class C {
   ...
   public void doSomething(){
        Y y1 = new Y(){ public perform(){...} };
        y1.perform();
   }
}
```

- *anonymous classes* sind unbenannte, lokale Klassen
- Anonyme Klassen können nur einmal instanziiert werden und haben keinen Konstruktor
- Wenn die genannten Bedingungen erfüllt sind und die Klassendefinition kurz ist, erhöhen anonymer Klassen die Lesbarkeit des Codes (kürzer, klarer)

3.2 Programmiermethodik und Softwaretechnik

Die Erstellung großer Programmsysteme ist schwierig und kann daher nicht in einer ad hoc-Manier bewältigt werden, vielmehr ist planvolles Vorgehen nötig. Dieser Abschnitt führt kurz in die Problematik ein.

3.2.1 Komplexität bei der Softwareerstellung

Faktoren sind:

- *Problemkomplexität*: komplexe Probleme (Reaktorsteuerung etc.) *Verstehen, Beschreiben*
- *Lösungskomplexität*: komplexe Lösungen (und Programme)
  Aufbau, Architektur
- *Teamkomplexität*: viele u.U. sehr unterschiedliche Programmierer
  Koordination, Projektmanagement

3.2.2 Phasenmodell der SW-Erstellung

(auch „Software Life-Cycle“) beschreibt eine Möglichkeit, die Softwareentwicklung in Phasen zu gliedern.

3.2.2.1 Analyse und Modellierung

„*was* ist zu tun?“

- **Analyse** des Problembereichs in einem Analysemodell, präzise Beschreibung der Anforderungen an das System in einer Anforderungsdefinition (auch: Pflichtenheft)
  -> „*Problem* verstehen und beschreiben“
  - Beispiel: Brücke über einen Fluß bauen

    Modellierung der Umgebung/Umwelt

    Formulierung der Anforderungen: Auto-, Eisenbahn o. Fußgängerbrücke?, zulässige Gewichte?, klappbar?, erdbebensicher?, sturmfest? etc.
  - Software: Unterscheidung zwischen funktionalen und nicht-funktionalen Anforderungen
- **Modellierung** der Lösung -> „*Lösung* abstrakt beschreiben“
  - Brücke: Modell in kleinem Maßstab mit allen benötigten Komponenten, bei Klappbrücke evtl. funktionsfähig
  - Software: häufig Diagramme und *funktionale Spezifikation* (s. 3.2.3) des Systems: „welche Komponente tut unter welchen Bedingungen was“, Angabe von Bedingung und Effekt von Operationen, möglicherweise anhand formaler Modelle.

- Ist eine Spezifikation konsistent, widerspruchsfrei, korrekt, vollständig?

3.2.2.2 Entwurf und Implementierung

„*wie* ist das System zu bauen?“

- **Entwurf** des Systems, evtl. durch Verfeinerung des Modells
  - Brücke: Zugmotor jetzt konkret: wie stark muß der Motor sein, gibt es so was schon auf dem Markt, müssen wir einen entwickeln oder in Auftrag geben. Was für Stahlträger verwenden wir? Usw. usw.
  - Software: Entwurf von Klassen mit bestimmten Algorithmen, jetzt auch unter Performance-Gesichtspunkten
- **Implementierung**: Umsetzung des Entwurfs, Codierung in einer oder mehreren Programmiersprachen

3.2.2.3 Testen der Komponenten

- in Einzelteilen

3.2.2.4 Systemintegration- und Test,

- Zusammenfügen und Testen des Ganzen, „Aufbau“ beim Kunden

3.2.2.5 Wartung, Pflege

- Fehlerbeseitigung und Weiterentwicklung

3.2.3 Funktionale Spezifikation

= genaue Bestimmung der Funktion eines Systems/Bausteins unter Abstraktion von Details der Realisierung.

- **Sinn**: sowohl zur Festlegung des Modells wie auch zur Dokumentation -> Wartung, Wiederverwendung
- Beispiel: Wecker
  Funktionen: Zeit anzeigen, zu bestimmten Zeiten Signal geben -> wie kann man diese Funktionen nun genau beschreiben (spezifizieren)?
  - Formal: auf einem formalen (mathemat.) Modell basierend
    -> formale Spezifikationssprachen
  - (umgangs-)sprachlich:
    „Ein Wecker zeigt stets die aktuelle Zeit an. Er gibt ein Wecksignal, wenn die aktuelle Zeit gleich der eingestellten Weckzeit ist und die Weckfunktion aktiv ist.“
  - halb-formal, z.B.:

```
Wecker [
            aktuelleZeit, weckzeit: ZEIT;
            weckfkt, wecksignal: {aktiv, inaktiv};
```

```
]
op zeitAnzeigen liefert wert: Zeit
                Voraussetzung: -
                Effekt: wert = aktuelleZeit

op wecken
                Voraussetzung: weckfkt = aktiv und
                                aktuelleZeit = weckzeit
                Effekt: wecksignal = aktiv

op weckfunktionAktivieren
                Voraussetzung: -
                Effekt: weckfunktion = aktiv
op weckzeitEinstellen benötigt weckZeit: Zeit
                Voraussetzung: gültig(weckZeit)
                Effekt: weckzeit = weckZeit

op aktuelleZeitEinstellen benötigt aktZeit: Zeit
                Voraussetzung: gültig(aktZeit)
                Effekt: aktuelleZeit = aktZeit
```

3.2.4 Spezifikation und Java

Kann man in Java spezifizieren? Wenn ja, wie?

- Betrachte Implementierung als Spezifikation:
  überspezifiziert, nicht abstrakt genug, zu viele irrelevante Details, verwirrend
- Betrachte Interfaces als Spezifikation:
  unterspezifiziert, Semantik nur unzulänglich durch Methodennamen und Argumenttypen angedeutet.

⇒ Möglichkeit: reichere Interfaces mit informeller, sprachlicher Spezifikation in Kommentaren an!

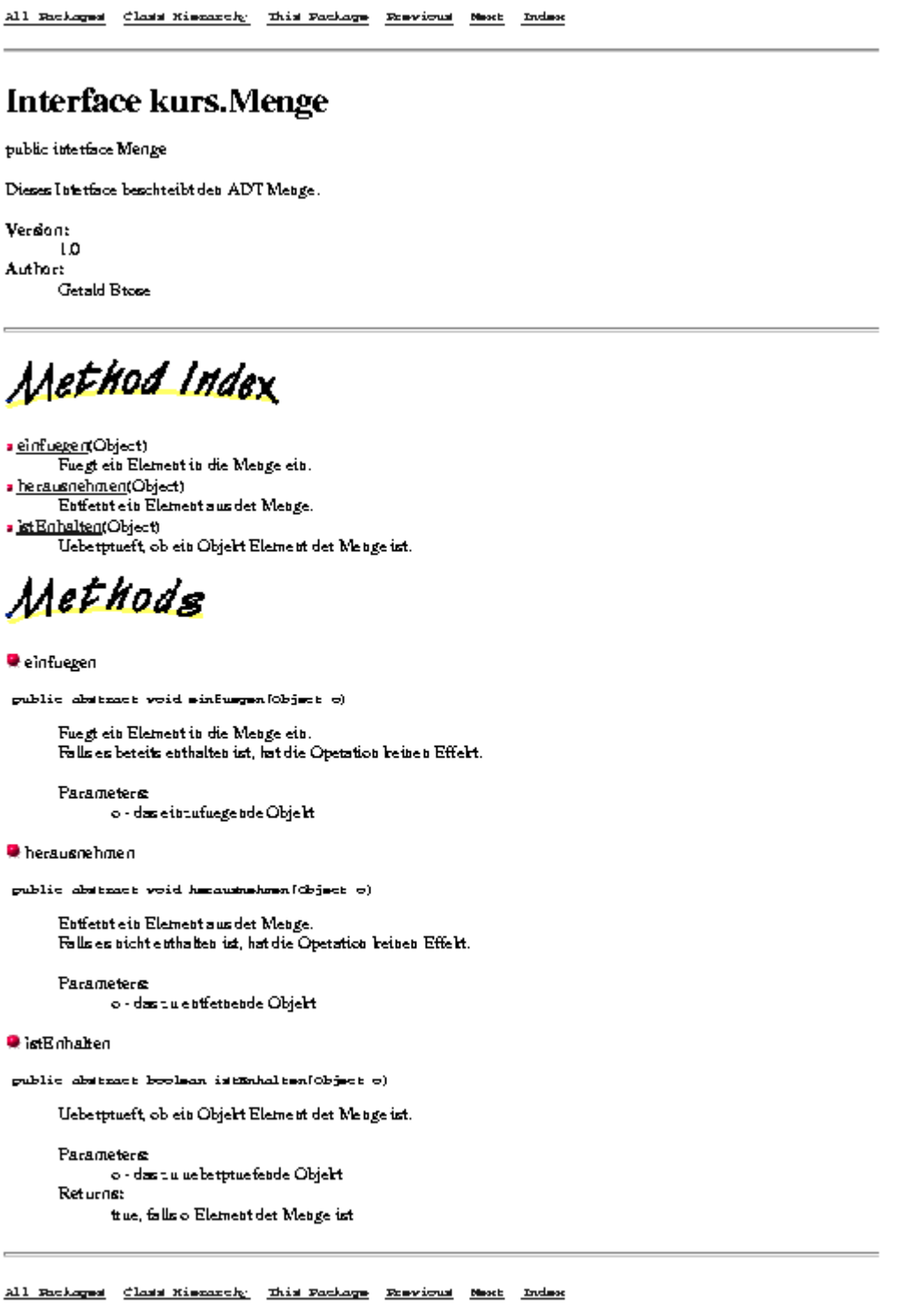
3.2.4.1 Dokumentation mit javadoc

- **javadoc** ist ein Generator für Klassendokumentation in HTML-Format, der Quelltext parsiert und dabei bestimmte Kommentare erkennt.
- javadoc-Kommentare haben folgendes Format:
  ```
  /** javadoc-Kommentar */
  ```
- erkennt bestimmte Markierungen („tags“):
  - für Klassen und Interfaces: `@author, @version`
  - für Methoden: `@param, @return, @exception`
  - leider keine vordefinierten Tags für Voraussetzung und Effekt, ist aber erweiterbar!

- Beispiel:

```
package kurs;
/**
 * Dieses Interface beschreibt den ADT Menge.
 * @author Gerald Brose
 * @version 1.0
 */

public interface Menge {

     /**
      * Fuegt ein Element in die Menge ein. <br>
      * Falls es bereits enthalten ist, hat die Operation
      * keinen Effekt.
      * @param o das einzufuegende Objekt <br>
      */
     void einfuegen(Object o);

     /**
      * Entfernt ein Element aus der Menge. <br>Falls es
      * nicht enthalten ist, hat die Operation keinen
      * Effekt.
      * @param o das zu entfernende Objekt <br>
      */
     void herausnehmen(Object o);

     /**
      * Ueberprueft, ob ein Objekt Element der Menge ist.
      * @param o das zu ueberpruefende Objekt
      * @return true, falls o Element der Menge ist
      */
     boolean istEnhalten(Object o);
}
```

- `javadoc -author -version Menge.java`
  erzeugt HTML-Dokument kurs.Menge.html



All Packages Class Hierarchy This Package Previous Next Index

Interface kurs.Menge

public interface Menge

Dieses Interface beschreibt den ADT Menge.

Version:
1.0
Author:
Gerald Brose

Method Index

- einfuegen(Object)
  Fuegt ein Element in die Menge ein.
- herausnehmen(Object)
  Entfernt ein Element aus der Menge.
- istEnthalten(Object)
  Ueberprueft, ob ein Objekt Element der Menge ist.

Methods

einfuegen

```
public abstract void einfuegen(Object o)
```

Fuegt ein Element in die Menge ein.
Falls es bereits enthalten ist, hat die Operation keinen Effekt.

Parameters:
o - das einzufuegende Objekt

herausnehmen

```
public abstract void herausnehmen(Object o)
```

Entfernt ein Element aus der Menge.
Falls es nicht enthalten ist, hat die Operation keinen Effekt.

Parameters:
o - das zu entfernende Objekt

istEnthalten

```
public abstract boolean istEnthalten(Object o)
```

Ueberprueft, ob ein Objekt Element der Menge ist.

Parameters:
o - das zu ueberpruefende Objekt
Returns:
true, falls o Element der Menge ist

All Packages Class Hierarchy This Package Previous Next Index

1 of 1 08.10.98 1

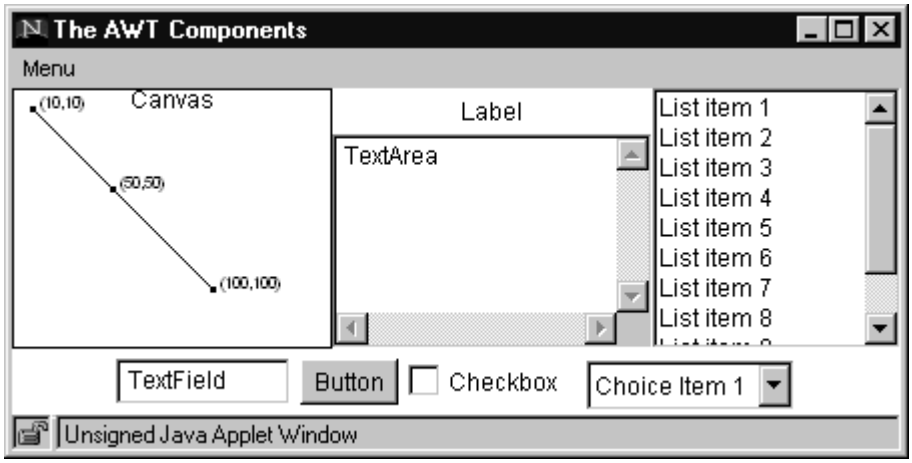
- die erste Zeile der Methodenbeschreibung erscheint zusätzlich im *method index* -> Stil: erster Satz beschreibt Effekt, dann Bedingungen.

3.3 GUI-Programmierung

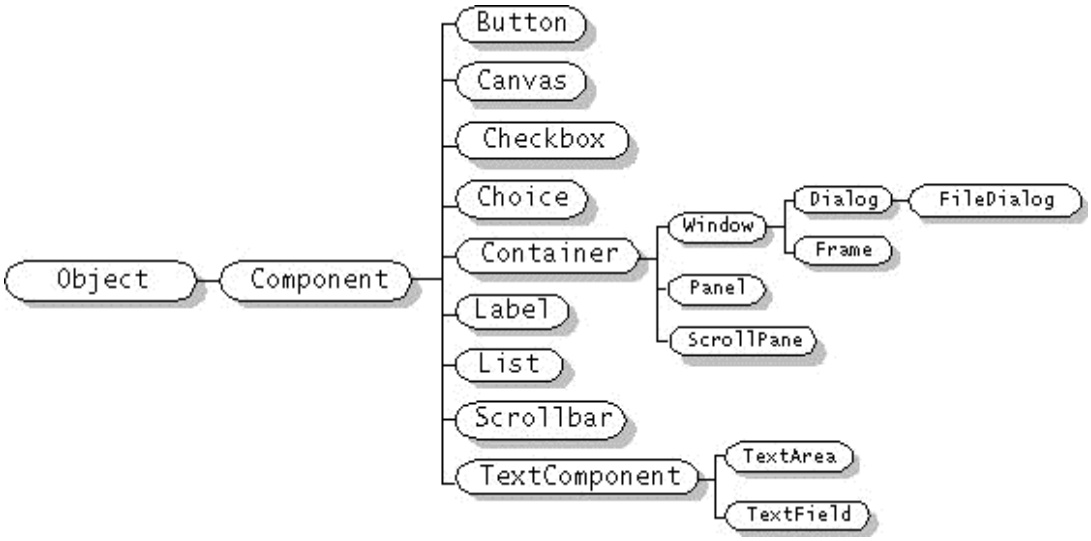
- Java GUI-Bibliothek: AWT (*Abstract Windowing Toolkit*)
- Ähnlich wie schon bei Streams wird Unabhängigkeit vom konkreten Betriebssystem und Fenstersystem angestrebt: Programme, die auf AWT basieren, sind im Idealfall plattformunabhängig.

3.3.1 Klassen im AWT

- GUIs bestehen in der Regel aus einem oder mehrere Fenstern (windows) mit weiteren graphischen Elementen wie Knöpfen (buttons), Rollbalken (scroll bars), Pull-down- und Pop-up-Menüs, Textfeldern, etc. wie in der folgenden Abbildung gezeigt.



  - Diese GUI-Elemente reagieren in der Regel auf Benutzereingaben über die Tastatur oder die Maus. Solche Benutzereingaben werden dem Programm in der Form von *Ereignissen* (*Events*) übergeben.
  - Das AWT stellt Klassen für alle gängigen GUI-Elemente zur Verfügung. Alle AWT-Klassen (bis auf Menüs) erben von `Component`. Die folgende Abbildung stellt die Vererbungsbeziehung der wichtigsten AWT-Klassen dar.



- Die Oberklasse `Component` stellt den Teil der Funktionalität von GUI-Elementen bereit, der allen solchen Elementen gemeinsam ist:
  - Grundlegende Darstellung auf dem Bildschirm: Größe und Position auf dem Bildschirm, Farben, Fonts
  - Ereignisbehandlung für allgemeine Benutzerereignisse

3.3.1.1 Container

- *Container* sind solche Komponenten, die andere Komponenten enthalten können. Container haben Methoden zum Hinzufügen wie Entfernen von Komponenten sowie zur Weiterleitung bestimmter Ereignissen.
- Unterklassen der Containerklasse *Window* sind eigenständige Fenster, die nicht selbst in anderen Containern enthalten sind.
- *Frames* sind unabhängige Fenster, *Dialogs* sind Dialogfenster, die etwa zur Auswahl einer zu öffnenden Datei oder zur Bestätigung verwendet werden.

Beispiel 1: *Öffnen eines leeren Fensters auf dem Bildschirm*

```
package kurs;
import java.awt.*;

public class BaseFrame
     extends Frame
{
     public static Dimension minSize =
                                   new Dimension(300,200);

     public BaseFrame(String title) {
          super(title);
          setSize( minSize );
     }

     public BaseFrame()
     {
          setTitle( getClass().getName() );
          setSize( minSize );
     }

     public static void main(String [] argv) {
          BaseFrame f;
          if( argv.length == 1 )
               f = new BaseFrame( argv[0] );
          else
               f = new BaseFrame();
          f.show(); // anzeigen!
     }
}
```

- **Achtung**: Wenn für einen Frame nicht explizit eine Größe gesetzt wird, wird

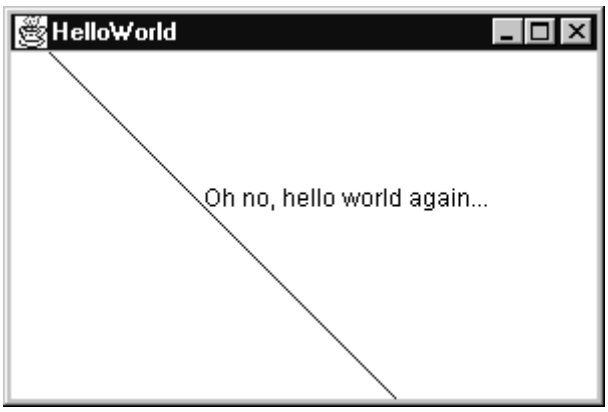
eine Standardgröße von 0 x 0 Pixeln verwendet, d.h. der Frame wird nicht angezeigt!

- Das Darstellen von Text oder das Zeichnen in einer Komponente geschieht durch das Überschreiben der Methode `paint()`, die von `Component` geerbt wird:

  Beispiel 2: *Ausgabe eines Strings und Zeichnen einer Linie*

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

public class HelloWorldFrame
     extends CloseableFrame
{
     public HelloWorldFrame()
     {
          super("HelloWorld");
          setBackground(Color.white);
     }
     public void paint(Graphics g )
     {
          g.drawString("Oh no, hello world again...",
100,100);
          g.drawLine(0,0,200,200);
     }
     public static void main(String [] argv)
     {
          HelloWorldFrame f = new HelloWorldFrame();
          f.show();
     }
}
```

Ausgabe:



- Die Klasse `Graphics` bietet zahlreiche Methoden zum Zeichnen und Schreiben.

- Die Methode `paint()` wird jedesmal aufgerufen, wenn die Komponente ein „Update“-Ereignis empfängt (z.B. nach einer Änderung der Fenstergröße oder dem Öffnen, nachdem das Fenster „ikonisiert“ war).

3.3.1.1.1 Hinzufügen von Komponenten zu Containern

- Das Hinzufügen einer GUI-Komponente zu einem Container geschieht mit der Methode `add()` des betreffenden Containers.

- **Beispiel 3:** *ein HelloWorld-Fenster mit einem Textfeld und einem Button*

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

public class HelloWorldButton extends HelloWorldFrame {
     TextField textField;
     String text = "Still there: HelloWorld!";
     Button button;

     public HelloWorldButton() {
          super("HelloWorld with Button");
          setLayout( new FlowLayout());
          textField = new TextField( text);
          button = new Button("Clear");
          add(textField);
          add(button);
     }

     public void paint(Graphics g ) {
          textField.setText(text);
     }

     public static void main(String [] argv) {
          HelloWorldButton f = new HelloWorldButton();
          f.show();
     }
}
```

- Der im Textfeld dargestellte Text ist standardmäßig editierbar, allerdings wird er bei jedem Aufruf der `paint`-Methode – also bei jedem Neuzeichnen – wieder auf den Ausgangstext gesetzt.

- Die Anordnung der Komponenten in einem Container wird durch die Wahl eines `LayoutManagers` bestimmt:
  - Als Standard wird `BorderLayout` verwendet. Bei diesem Layout wird die Position einer Komponente im Container in Himmelsrichtungen angegeben. Neben „North“, „South“, „West“, und „East“ gibt es noch „Center“.
    Achtung: bei `BorderLayout` muß `add()` mit einem zusätzlichen String-Parameter aufgerufen werden, der die Position bezeichnet, ansonsten erscheinen nicht alle Komponenten auf dem Bildschirm!
  - `FlowLayout` bedeutet „alle hintereinander“.
  - Zu weiteren Details von `LayoutManagern` und Komponentenlayout siehe z.B. „The Java Tutorial“ von Sun.
- Der Button im Beispiel kann zwar angeklickt werden, hat aber noch keine Funktion. Wie die Reaktion einer Komponente auf ein Ereignis programmiert werden kann, wird in 3.3.2 dargestellt.

3.3.2 Benutzereingaben und Events

- Eine Benutzeraktion (mit der Maus oder der Tastatur) auf einem GUI-Element veranlaßt dieses, ein *Ereignis* bekanntzugeben: GUI-Elemente sind Ereignisquellen (*event sources*).
- Eine Reaktion auf solche Ereignisse erfolgt einem oder mehreren Objekten, die solche Ereignisse wahrnehmen, dem sog. *Event Listener*
- GUI-Elemente haben Methoden, mit denen man Objekte als *Listener* für bestimmte Ereignisklassen anmelden kann, z.B als `MouseListener`, `KeyListener` oder als `WindowListener`.
- Konkrete Listener für bestimmte Ereignisse müssen entsprechende Interfaces implementieren, z.B. das Interface `MouseListener`, das Methoden zur Ereignisbehandlung für Mausereignisse wie Drücken oder Loslassen der Maustasten vorsieht.
- **Beispiel**: ein *WindowListener*, der als Reaktion auf ein Schließ-Ereignis des Fensters das Programm beendet:

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

public class FirstWindowListener  implements
WindowListener {
     public void windowActivated(WindowEvent e) {}
     public void windowClosed(WindowEvent e) {}
     public void windowClosing(WindowEvent e) {
          System.exit(0);
     }
     public void windowDeactivated(WindowEvent e) {}
```

```
     public void windowDeiconified(WindowEvent e) {}
     public void windowIconified(WindowEvent e)  {}
     public void windowOpened(WindowEvent e) {}
}
```

- Die Klasse `FirstWindowListener` muß alle Methoden des Interface `WindowListener` implementieren. Da nur die Reaktion auf `windowClosing` interessant ist, werden für die anderen Methoden lediglich leere Rümpfe angegeben.
- Wird ein Objekt dieser Klasse bei einem Fenster als `WindowListener` angemeldet, so führt ein Schließen des Fensters zum Aufruf der Methode `windowClosing`, was nun die Beendigung des Programms zur Folge hat (der explizite Aufruf von `System.exit()`).
- **Beispiel**: Anmelden eines `WindowListeners` bei einem `Frame`.

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

public class CloseableFrame extends BaseFrame {

     public CloseableFrame(String title){
          super(title);
          addWindowListener( new FirstWindowListener());
     }

     public CloseableFrame(){
          super();
          addWindowListener( new FirstWindowListener());
     }

     public static void main(String [] argv){
          CloseableFrame f;
          if( argv.length == 1 )
               f = new CloseableFrame( argv[0] );
          else
               f = new CloseableFrame();
          f.show();
     }
}
```

- `CloseableFrame` reagiert nun wie erwartet auf das Klicken auf das Schließsymbol des Fensters.
- Java stellt als Basis für eigene Listener leere Implementierung aller Listener-Interfaces bereit – sogenannte *Adapterklassen*. Diese Adapter stellen für jede Methode eines Listener-Interfaces eine leere Methodenimplementierung bereit, so daß lediglich die benötigten Methoden in Unterklassen überschrieben werden müssen.
- Die entsprechen Adapterklasse für `WindowListener` heißt `WindowAdapter`. Damit läßt sich die Implementierung eines eigenen

Listeners stark vereinfachen:

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

public class SecondWindowListener extends WindowAdapter {
     public void windowClosing(WindowEvent e) {
          System.exit(0);
     }
}
```

- Noch knapper läßt sich die Definition und das Anmelden von eigenen Listenern für bestimmte Ereignisse bestimmter GUI-Elemente durch die Verwendung von *anonymous classes* (s. 3.1.4) formulieren:

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

public class CloseableFrame extends BaseFrame {

     public CloseableFrame() {
          super();
          addWindowListener( new WindowAdapter(){
               public void windowClosing(WindowEvent e)
                    {System.exit(0);} } );
     }

     public static void main(String [] argv) {
          CloseableFrame f;
          if( argv.length == 1 )
               f = new CloseableFrame( argv[0] );
          else
               f = new CloseableFrame();
          f.show();
     }
}
```

- Die Definition einer neuen (anonymen) Listener-Klasse als Unterklasse von `WindowAdapter`, die Erzeugung eines Objekts dieser Klasse und das Anmelden als Listener geschieht auf einer Zeile!

- Für komplexere GUIs wird typischerweise eine größere Anzahl verschiedener Listener für die diversen Ereignisse benötigt, was zu einer Großzahl kleiner Klassen führt. Ebenso typischerweise wird von jeder Listener-Klasse jeweils nur ein Objekt benötigt. Falls nur jeweils ein Objekt benötigt wird und dieses keinen Konstruktor benötigt, sind *anonymous classes* sind die bequemste und auch „lesbarste“ Lösung.

- Eine andere typische Situation ist im Beispiel `HelloWorldButton` zu sehen: Damit als Reaktion auf das Klicken des Clear-Buttons das Textfeld gelöscht werden kann, benötigt der zu definierende Listener Zugriff auf das Textfeld: entweder, die `HelloWorldButton`-Klasse selbst reagiert als `MouseListener/Adapter` auf das Ereignis, oder man definiert eine *member*

class (s. 3.1.2), deren Exemplar implizit mit dem `HelloWorldButton`-Objekt verbunden ist und daher Zugriff auf das Textfeld hat.

- Im konkreten Fall scheidet die erste Möglichkeit aus, da `HelloWorldButton` bereits eine Oberklasse hat und daher nicht mehr Unterklasse von `MouseAdapter` sein kann, daher wird im folgenden Beispiel eine member class `ClearButtonReactor` definiert, die direkt auf die `textField`-Variable des `HelloWorldButton`-Objekts zugreifen kann, das den `ClearButtonReactor` erzeugt.:

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

public class HelloWorldButton extends HelloWorldFrame {
   TextField textField;
   String text = "Still there: HelloWorld!";
   Button button;

   class ClearButtonReactor extends MouseAdapter {
        public void mouseReleased(MouseEvent e)
        {
             textField.setText("");
        }
   }

   public HelloWorldButton() {
        super("HelloWorld with Button");
        setLayout( new FlowLayout());
        textField = new TextField( text);
        button = new Button("Clear");
        button.addMouseListener( new ClearButtonReactor());
        add(textField);
        add(button);
   }

   public void paint(Graphics g ) {
        textField.setText(text);
   }
   public static void main(String [] argv) {
        HelloWorldButton f = new HelloWorldButton();
        f.show();
   }
}
```

3.4 Applets

Applets sind in Java geschriebene Mini-Anwendungen, die in anderen Programmen, die als „Appletviewer“ fungieren, ausgeführt werden – was besonders durch die Integration in Web-Browsern interessant ist, denn Applets können auf HTML-Seiten ablaufen.

3.4.1 Überblick

- Applets sind keine selbständigen Applikationen und haben keine `main()`-Methode. Sie steuern den Kontrollfluß nur eingeschränkt, vielmehr reagieren sie auf Anforderungen der Umgebung (des Browsers).
- Applets sind Unterklassen von `java.applet.Applet`, das wiederum eine Unterklasse von `Panel` ist. Applets erben daher indirekt von sowohl von `Container` wie von `Component` und haben die gleiche Ereignisbehandlung wie andere AWT-Klassen.
- Applets werde geladen, wenn ein Browser auf das HTML-Tag <APPLET> trifft. Als Beispiel hier der HTML-Text einer Seite, die ein HelloWorld-Applet enthält:

```
<TITLE>HelloWorldApplet</TITLE>
<CENTER>
<h1>HelloWorldApplet</h1>
</CENTER>
<HR>
<APPLET CODE=HelloWorldApplet.class WIDTH=350 HEIGHT=200>
</APPLET>
<HR>
```

- Applets laufen immer auf dem lokalen Rechner. Der Applet-Code kann über das lokale Dateisystem geladen werden, kommt aber typischerweise von einem entfernten Rechner. Der Parameter `CODE` des APPLET-Tags gibt an, welche Klasse zu laden ist und wird relativ zur Position (d.h. der URL) der betreffenden HTML-Seite verstanden, wenn nicht ein weiterer Parameter `CODEBASE` eine andere Codelokation festlegt.

3.4.2 Ein HelloWorld-Applet

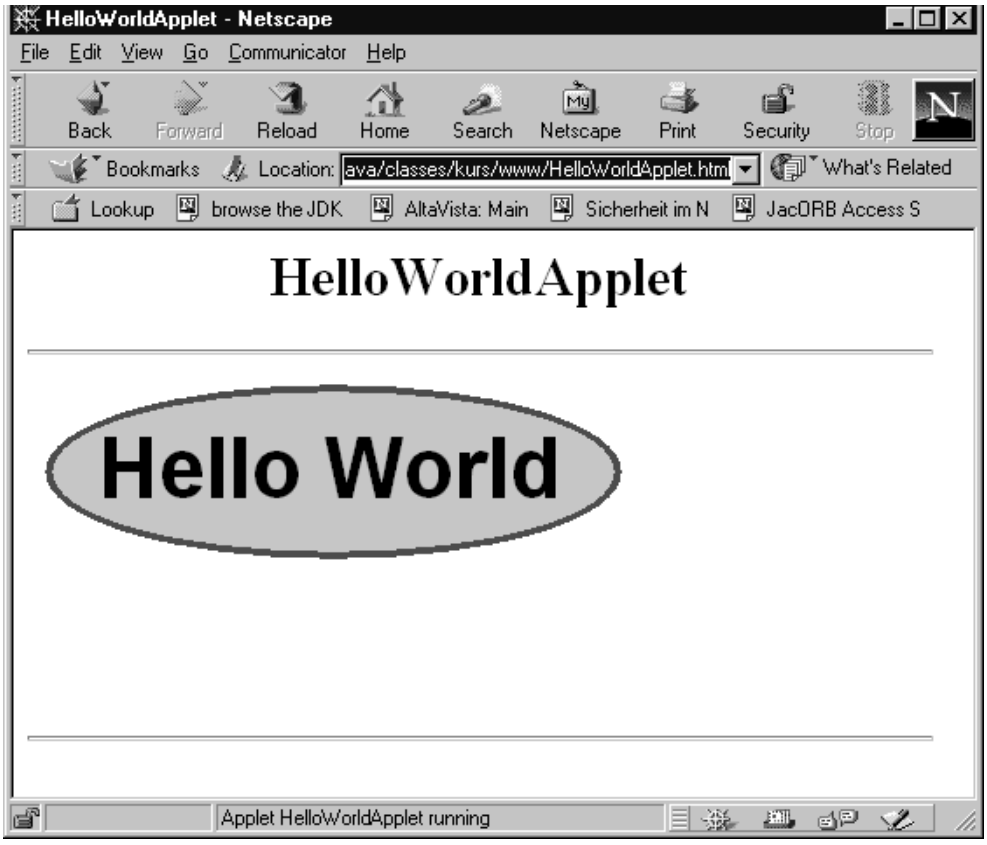
- Der folgende Code definiert eine Appletklasse, das zwei farbige Ovale übereinanderlegt und die bekannte Nachricht hineinschreibt:

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

public class HelloWorldApplet extends Applet  {
     private Font font;

     public void init(){
          font = new Font("Helvetica", Font.BOLD, 48);
     }
     public void paint(Graphics g)
```

```
    {
        g.setColor( Color.red );
        g.fillOval( 10,10,330,100);
        g.setColor(Color.pink);
        g.fillOval( 14,14,322,92 );
        g.setColor(Color.black);
        g.setFont(font);
        g.drawString("Hello World", 40,75);
    }
}
```

- Wenn das Applet von einem Browser geladen wird, ruft dieser zunächst als einmalige Initialisierung die `init()`-Methode auf. Achtung: für Applets werden keine Konstruktoren aufgerufen. Die Methode `paint()` wird wie schon bei AWT-Komponenten als Reaktion auf ein `update()` aufgerufen.

- Die entsprechende Anzeige im Browser sieht so aus:



- Die Position und Größe eines Applets in einer HTML-Seite wird durch die Parameter `HEIGHT` und `WIDTH` des APPLET-Tags bestimmt:

```
<APPLET CODE=HelloWorldApplet.class WIDTH=350 HEIGHT=200>
</APPLET>
```

3.4.3 Einschränkungen für Applets

3.4.3.1 Sicherheitsrestriktionen

- Da Applet-Code in der Regel nicht nur selbstgeschrieben ist, sondern auch (zum Teil unbemerkt) von entfernten und nicht vertrauenswürdigen Rechnern heruntergeladen wird, stellt eine unbeschränkte Ausführung ein erhebliches Sicherheitsrisiko dar.
- Um zu verhindern, daß Applets beliebigen Schaden auf dem lokalen Rechner anrichten können, werden ihre Ausführungsrechte drastisch beschnitten. Die genauen Einschränkungen hängen vom jeweiligen Browser ab, generell dürfen Applets aber nicht:
  - Im lokalen Dateisystem lesen oder schreiben
  - Netzwerkverbindungen öffnen (außer zu dem Rechner, von dem sie geladen wurden) oder auf eingehende Verbindungen warten
  - Neue Prozesse starten (durch die Methode `Runtime.exec()`)
  - Dynamisch *native code* laden
  - Einen Druckauftrag abgeben, auf das Clipboard oder die System-Eventqueue zugreifen
  - Auf beliebige *system properties* zugreifen
  - Auf Threads außerhalb der eigenen Thread-Gruppe zugreifen
  - Eigene `ClassLoader` einrichten
- Durch das *digitale Signieren* und entsprechende Sicherheitspolitiken des Browsers lassen sich solche Einschränkungen lockern, auch gelten im Regelfall lockerere Regeln für Applets, die über das lokale Dateisystem geladen werden.

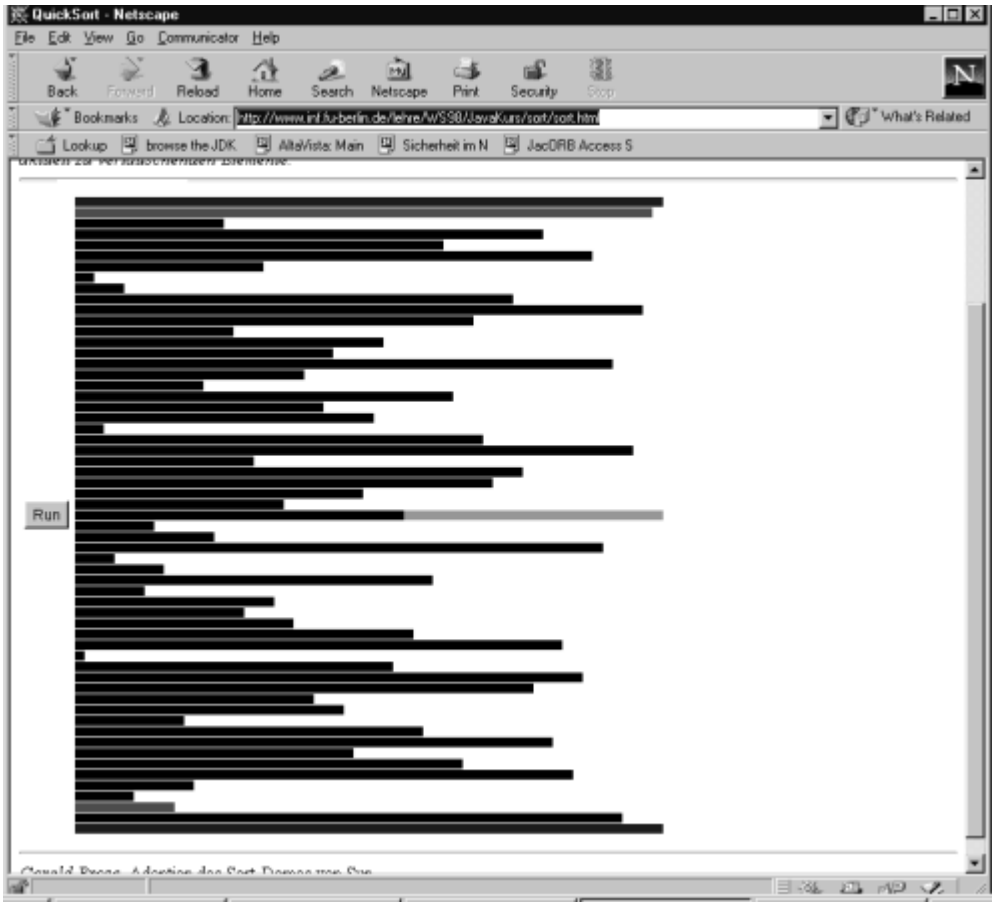
3.4.3.2 GUI-Einschränkung

- Da Applets kein eigenes Fenster verwalten, sondern eine Unterklasse von `Panel` sind, können sie keinen eigenen Menübalken haben. Bei der Adaptierung bestehender Programme mit graphischer Oberfläche muß man darauf achten, daß Menübalken geeignet durch Buttons o.ä. nachgemacht werden müssen.
- Das Standard-Layout für Applets ist nicht, wie bei Frames, `BorderLayout`, sondern `FlowLayout`, d.h. Anwendungen, deren graphische Oberfläche auf einem impliziten `BorderLayout` beruht, müssen dieses Layout explizit setzen.
- Wenn Applets eigene Fenster öffnen, werden diese in der Regel vom Browser mit einer Markierung versehen, um Verwechslungen mit den Fenstern lokaler Anwendungen zu vermeiden (Sicherheitsrisiko!). Ein Browser kann ein solches Fenster z.B. mit einer Fußzeile mit der Meldung „Unsigned Java Applet Window“ versehen.

3.4.3.3 Beispiel: QuickSort-Applet

Das QuickSort-Applet zeigt neben den bereits bekannten Aufgaben der Anordnung von GUI-Komponenten und der Ereignisbehandlung zwei weitere wesentliche Techniken bei der Programmierung von Applets: einerseits verwaltet es eigene Threads (hier nicht erklärt), andererseits verwendet es eine Technik namens *Double Buffering*, um ein Bildschirmflackern beim Neuzeichnen von Graphik zu vermeiden. Die `init()`-Methode zeigt außerdem, wie die Werte von Parametern des APPLET-Tags ausgelesen werden können.

Die Bildschirmausgabe des QuickSort-Applets sieht so aus (im Original farbig):



Der Text der HTML-Seite, die das Applet lädt:

```
<p><i>Der Button schaltet zwischen Single-Step und
Animationsmodus um. Die blauen Markierungen geben die Grenzen
des Bereichs an, in dem aktuell sortiert wird, die grüne
Markierung zeigt den Mittelwert, die roten Markierungen sind
für die aktuell zu vertauschenden Elemente.</i>
<hr>
<APPLET CODE=SortItem.class width=540 height=540>
<PARAM NAME=mode VALUE="single_step"></APPLET>
<hr>
<i>Gerald Brose</i>, Adaption des Sort-Demos von Sun
```

Die Appletklasse `SortItem` definiert mehrere *inner classes*. `LocalCanvas` ist eine Unterklasse von `Canvas` und zuständig für die graphische Darstellung des Sortiervorgangs. Um Bildflackern zu vermeiden, werden Zeichenoperationen nicht direkt auf dem `Graphics`-Object vorgenommen, das am Bildschirm aktuell angezeigt wird. Statt dessen wird auf einem „off-screen“ Objekt gezeichnet, das schließlich in einer einzigen Zeichenoperation auf dem Bildschirm ausgegeben wird.

Die *member class* `ButtonController` behandelt die Mausereignisse auf dem Button und schaltet beim Eintreffen eines solchen Ereignisses zwischen Einzelschritt- und Animationsmodus um.

Die `init()`-Mehode greift durch Aufruf von `getParameter()` auf den String-Wert des VALUE-Feldes zu, das Teil des entsprechend PARAM-Tags ist.

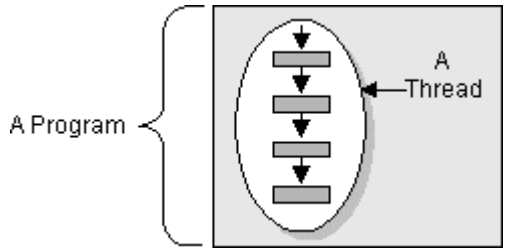
```
import java.awt.*;
import java.awt.event.*;

public class SortItem
     extends java.applet.Applet
     implements Runnable {

     ...
    private Button finishButton;
    private LocalCanvas sortCanvas;
    private Dimension size;

    /** The array that is being sorted. */
    int arr[];
     ...

    /** Whether this applet runs in single stepping mode. */
    boolean single_stepping;

    /** The sorting algorithm  */
    QSortAlgorithm algorithm;

     public class LocalCanvas extends Canvas {
          Dimension size;
          Image offscreen;

          public LocalCanvas(Dimension s){
               size = s;
          }

          public Dimension getPreferredSize(){
               return size;
          }

          public Dimension getMinimumSize(){
               return size;
          }
          /**
           * Update without erasing the background.
           */
          public void update(Graphics g){
```

```
            paint(g);
        }

        /**
         * Double buffering.
         */
        public void paint(Graphics g) {
          if(offscreen == null) {
             offscreen = createImage(getSize().width,
                                     getSize().height);
          }
          Graphics og = offscreen.getGraphics();
          paintOff(og);
          g.drawImage(offscreen, 0, 0, null);
          og.dispose();
        }

        /**
         * Paint the array of numbers as a list
         * of horizontal lines of varying lenghts.
         */
        public void paintOff(Graphics g) {
            int a[] = arr;
            int height = (getSize().height/9)*9;
            int y =  height -8;

            for (int i = a.length; --i >= 0; y -= 9) {
                 // Draw new lines
                  g.setColor(Color.black);
                  g.fillRect(0, y, arr[i], 8);
                 // Erase old lines
                  g.setColor(getBackground());
                  g.fillRect(arr[i], y, getSize().width, 8)
            }

            if (h1 >= 0){
             g.setColor(Color.blue);
             y = h1 * 9 + 1;
             g.fillRect(arr[h1], y, getSize().width, 8);
            }
              ...
        }
    }

    /**
    * Initialize the applet.
    */
    public void init()
    {
         String mode = getParameter("mode");
         single_stepping = mode.equals("single_step");
         finishButton = new Button("Run");
         finishButton.addMouseListener(
                              new ButtonController());
         size = getSize();
```

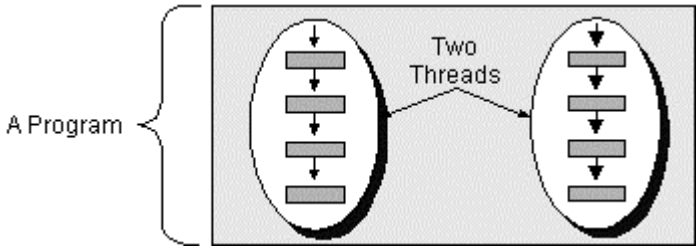
```
          size.setSize( size.width-50, size.height-9);
          sortCanvas = new LocalCanvas(size);
          sortCanvas.addMouseListener(new ClockStarter());
          sortCanvas.setBackground(Color.white);
          add(finishButton);
          add(sortCanvas);
     }

     public void update(Graphics g){
          sortCanvas.repaint();
     }

     class ButtonController extends MouseAdapter {
          public void mouseReleased(MouseEvent e){
               single_stepping = !(single_stepping);
               if( !single_stepping )
                    finishButton.setLabel("Stop");
               else
                    finishButton.setLabel("Run");
               ...
          }
     }
}
```

3.5 Threads

Die bisher betrachteten Java-Programme liefen *sequentiell* ab, d.h. Anweisungen wurden schrittweise und strikt nacheinander abgearbeitet – es gab genau einen Kontrollfluß (Kontrollfaden, engl. *Thread*) durch das Programm.



Java erlaubt jedoch auch die *nebenläufige* (engl. concurrent) Ausführung eines Programms durch mehrere solcher Threads, so daß mehrere Aktivitäten (quasi-) gleichzeitig ausgeführt werden können. Man spricht dann von einem nebenläufigen Programm (*concurrent* oder *multi-threaded program*).



3.5.1 Nebenläufigkeit – eine Einführung

Die folgende Liste gibt einige Beispiele für den Einsatz von Nebenläufigkeit:

- *(echte) Parallelität*: auf einer Maschine mit mehreren Prozessoren kann ein Programm i.d.R. schneller abgearbeitet werden, wenn unabhängige Teile gleichzeitig von mehreren Prozessoren bearbeitet werden (z.B. Multiplikation großer Matrizen)
- *reaktive Programme*: durch Nebenläufigkeit kann ein Programm wieder auf Benutzereingaben reagieren, obwohl noch eine andere Aktivität ausgeführt wird (z.B. eine Datenübertragung in einem WWW-Browser)
- *autonome Aktivitäten*: manche Aktivitäten sollen unabhängig vom Rest der Programmausführung stattfinden (z.B. ein Timer, der einen blockierenden Aufruf nach Ablauf eines Zeitintervalls abbricht).
- *asynchrone Aufrufe*: um bei einem Aufruf nicht auf das Ergebnis warten zu müssen, kann der Aufruf nebenläufig geschehen, während der Hauptthread mit der Arbeit fortfährt.

Nebenläufigkeit bedeutet aber auch zusätzliche Komplexität und Fehlermöglich-keiten. Folgende Nachteile können beim Einsatz von Nebenläufigkeit auftreten:

- *Korrektheit*: sind nebenläufige Aktivitäten nicht unabhängig, so können z.B. bei gleichzeitigem schreibenden Zugriff zweier Threads auf ein Objekt inkonsistente Zustände auftreten, wenn keine geeigneten Synchronisations-mechanismen verwendet werden. Solche Fehler sind schwer zu finden, und Synchronisation verursacht zusätzlichen Aufwand.
- *Verklemmungen* (engl. deadlock): zwei Aktivitäten können verklemmen, wenn beide darauf warten, daß die jeweils andere ein bestimmtes Ereignis auslöst, z.B. eine benötigte Ressource freigibt
- *Nichtdeterminismus*: die Ausführung nebenläufiger Aktivitäten kann beliebig verschränkt geschehen, so daß sie bei verschiedenen Programmläufen u.U. verschieden sein kann, was die Vorhersagbarkeit und das Verständnis erschwert.
- *Aufwand*: die Erzeugung und Verwaltung nebenläufiger Aktivitäten bedeutet zusätzlichen Aufwand.

3.5.2 Nebenläufigkeit in Java: Threads

- Nebenläufigkeit wird in Java durch das Erzeugen und Starten zusätzlicher Threads eingeführt. Ein Java-Thread ist ein Exemplar einer Unterklasse von `java.lang.Thread`.

Beispiel:

```
public class PingPong extends Thread {
   String word;
   int delay;

   PingPong( String word, int delay ){
        this.word = word;
        this.delay = delay;
   }

   public void run(){
        try {
             for(;;){
                  System.out.print( word + " " );
                  sleep( delay );
             }
        }
        catch (InterruptedException e){
             return;
        }
   }

   public static void main(String [] argv){
        new PingPong("PING", 33 ).start();
        new PingPong("pong", 100 ).start();
```

```
    }
}
```

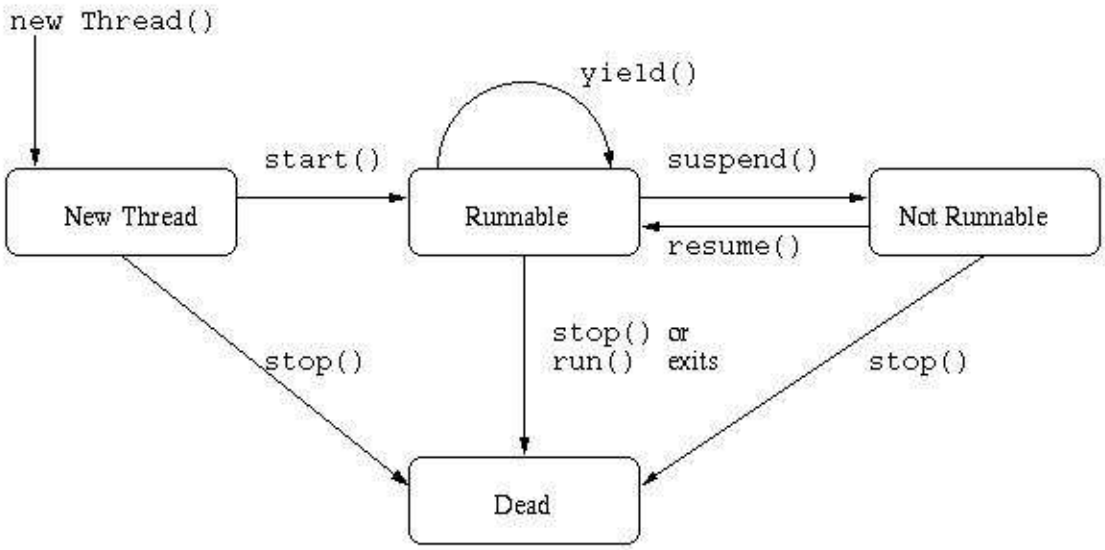
Ausgabe:

- `PING pong PING PING pong PING PING PING pong PING PING PING pong PING PING PING pong PING PING PING pong PING PING PING pong PING PING PING pong PING PING PING pong PING PING PING`

- `sleep()` ist eine Methode der Klasse `Thread` und hält den aufrufenden Thread für die angegebene Anzahl Millisekunden an. `sleep()` kann aber unterbrochen werden, falls auf dem der schlafenden Thread die Methode `interrupt()` aufgerufen wird. Dann endet `sleep()` mit einer `InterruptedException`.

3.5.2.1 Der Lebenszyklus von Threads

- Die folgende Abbildung zeigt alle möglichen Zustände eines Threads (von denen jedoch nicht alle hier erklärt werden):



- Ein Thread wird nach seiner Erzeugung durch den Aufruf `start()` gestartet. Dies veranlaßt, daß die `run()`-Methode des Threads ausgeführt wird. Der Aufruf von `start()` kann auch im Konstruktor der Thread-Klasse stehen, so daß der Thread gleich bei seiner Erzeugung gestartet wird.

- Ein Thread wird beendet, wenn

  - Die Ausführung aus der `run`-Methode zurückkehrt,

  - Die Methode `stop()` aufgerufen wird.

- Die Ausführung eines Threads kann durch Aufruf von `suspend()` an beliebiger Stelle angehalten und später mit `resume()` wieder fortgesetzt werden.

3.5.2.2 Thread vs. Runnable

- Wenn eine Klasse bereits eine Oberklasse hat, kann sie wegen der Einfachvererbung in Java nicht mehr von `Thread` erben. Für solche Fälle gibt es das Interface `Runnable`, das eine solche Klasse dann implementieren kann:

```
public class PingPong implements Runnable{
   String word;
   int delay;

   PingPong( String word, int delay ){
        this.word = word;
        this.delay = delay;
   }

   public void run(){
        try {
             for(;;){
                  System.out.print( word + " " );
                  Thread.currentThread().sleep( delay );
             }
        }
        catch (InterruptedException e){
             return;
        }
   }

   public static void main(String [] argv){
        new Thread(new PingPong("PING", 33 )).start();
        new Thread(new PingPong("pong", 100 )).start();
   }
}
```

- Da `PingPong` keine Unterklasse von Thread ist, kann die Methode `sleep()` nicht mehr direkt aufgerufen werden. Die statische Methode `currentThread()` der Klasse `Thread` liefert das benötigte Thread-Objekt für den aufrufenden Thread.

3.5.3 Sperrsynchronisation

- Threads müssen synchronisiert werden, wenn sog. *race conditions* drohen, d.h. Umstände, in denen der (quasi-)gleichzeitige Zugriff mehrerer Threads auf eine gemeinsame Variable zu einem „Wettrennen“ wird, in dem unterschiedliche Zugriffsreihenfolgen zu unterschiedlichen Ergebnissen führen.

- Angenommen, Konto-Objekte werden durch die Klasse `Account` wie folgt definiert:

```
public class Account {
   private int balance;

   public int getBalance(){
        return balance;
   }
```

```
   public void setBalance(int amount){
        balance = amount;
   }
}
```

Folgendes Szenario führt dann zu inkonsistenten Zuständen aufgrund von *race conditions*:

Thread 1:	**Thread 2**:
`int a1 = acc.getBalance();`	
	`int a2 = acc.getBalance();`
`a1 += deposit;`	
	`a2 += deposit;`
`acc.setBalance(a1);`	
	`acc.setBalance(a2);`

- Nach dem Ende dieser beiden Aktivitäten spiegelt der Kontostand lediglich eine (die letzte) der zwei Überweisungsvorgänge wider, die andere ist verloren!

3.5.3.1 Kritische Abschnitte

- Der gleichzeitige, lesende und schreibende Zugriff auf einen solchen *kritischen Abschnitt* muß daher durch gegenseitigen Ausschluß (*mutual exclusion*) verhindert werden.

Thread 1:	**Thread 2:**
`int a1 = acc.getBalance();`	
`a1 += deposit;`	
`acc.setBalance(a1);`	
	`int a2 = acc.getBalance();`
	`a2 += deposit;`
	`acc.setBalance(a2);`

- Achtung: es genügt nicht, lediglich das gleichzeitige Lesen oder das gleichzeitige Schreiben zu verhindern, sondern die ganze Transaktion bildet einen kritischen Abschnitt.
- Der gegenseitige Ausschluß wird in Java erreicht, indem ein Thread, der in einen kritischen Abschnitt eintreten will, vorher ein *Lock* für das Objekt er-

werben muß. Beim Verlassen des kritischen Abschnitts wird dieses *Lock* wieder freigegeben.

- Locks sind Java-Objekten zugeordnet: jedes Objekt hat ein eigenes Lock, zusätzlich gibt es weiteres ein Lock pro Klasse.
- Kritische Abschnitte werden in Java mit dem Schlüsselwort `synchronized` markiert. In *synchronisierten Methoden* wird implizit das Lock des aktuellen Objekts erworben und freigegeben

```
public class Account {
   private int balance;

   public synchronized int getBalance(){
        return balance;
   }

   public synchronized void deposit(int amount){
        balance += amount;
   }
}
```

- Durch die angegeben Synchronisierung in `Account` wird verhindert, daß ein Thread den Kontostand ändert, während ein anderer das gleiche versucht bzw. auch nur den Kontostand liest.
- Die angegebene Synchronisierung unterbindet allerdings auch nebenläufige Zugriffe, die ungefährlich sind, im Beispiel das gleichzeitige Lesen durch Aufruf von `getBalance()`.
- Die explizite Angabe, welches Lock (bzw. das Lock welchen Objekts) für die Synchronisation eines kritischen Abschnitts verwendet werden soll, ist bei *synchronized Statements* möglich:

```
public class Account {
   private int balance;
   private Object theLock = new Object();

   public int getBalance(){
        synchronized( theLock ){
             return balance;
        }
   }

   public void deposit(int amount){
        synchronized( theLock ){
             balance += amount;
        }
   }
}
```

3.5.4 Weitere Synchronisationsmöglichkeiten: Ereignisse, Join

Threads können außer über Sperrsynchronisation auch über das Auslösen von und das Warten auf Ereignisse koordiniert werden:

- der Aufruf der Methode `wait()` auf einem beliebigen Objekt blockiert den aufrufenden Thread, bis ein Ereignis eintritt.
- der Aufruf der Methode `notify()` bzw. `notifyAll()` auf demselben Objekt durch einen anderen Thread löst ein Ereignis aus und weckt den oder die wartenden Threads
- Der Aufruf von `wait()` oder `notify()` muß entweder in einer `synchronized` Methode des Objekts stattfinden, auf dem `wait/notify` aufgerufen wird, oder in einem `synchronized` statement, der ebenso auf diesem Objekt synchronisiert:

Thread A:

```
synchronized(o){
     o.wait();
}
```

Thread B:

```
synchronized(o){
     o.notify();
}
```

Eine weitere Möglichkeit, um Threads zu synchronisieren, ist das explizite Warten auf die Beendigung eines Threads durch den Aufruf von `join()`:

```
Thread t = new MyThread(x,y,z);
t.start();
doSomething();
t.join();
```

Der Aufruf von `t.join()` blockiert den aufrufenden Thread solange, bis `t` beendet wurde.

3.5.5 Threads in Applets: Das QuickSort-Applet

Das Applet-Beispiel aus 3.4.3.3 enthielt ursprünglich auch die Erzeugung eines eigenen Threads und war nur aus didaktischen Gründen verkürzt dargestellt. Der Sinn des Einsatzes von Threads in diesem Applet ist es, die Sortierung des Feldes nicht durchgängig durchzuführen, solange das Applet aktiv ist, sondern erst auf ein bestimmtes Ereignis hin, nämlich das Klicken in die Zeichenfläche. Als Reaktion auf dieses Ereignis wird ein eigener Thread gestartet, der im Hintergrund die Sortierung vornimmt.

Wäre die QuickSort-Berechnung selbst Teil der Ereignisbehandlung, dann wäre es während eines Animationslaufes nicht möglich, auf weitere Ereignisse zu reagieren und so die Animation wieder anzuhalten, um in den Einzelschrittmodus zurückzuschalten. Man kann daraus eine allgemeine Regel ableiten: um Applets *reaktiv* zu machen, sollte die Behandlung von Ereignissen möglichst kurz sein. Sind längere Berechnungen notwendig, sollte man

dafür eigene Threads starten.

Im konkreten Fall implementiert das Applet selbst das Interface `Runnable` und hat eine Instanzvariable vom Typ `Thread`:

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

public class SortItem
     extends java.applet.Applet
     implements Runnable
{

    /** The thread that is sorting (or null). */
    private Thread sortThread;
```

In der `init()`-Methode des Applets wird ein Objekt der *member class* `ClockStarter` als `MouseListener` für die lokale Zeichenfläche vereinbart:

```
public void init() {     ...
     sortCanvas = new LocalCanvas(size);
     sortCanvas.addMouseListener(new ClockStarter());
     sortCanvas.setBackground(Color.white);
     add(finishButton);
     add(sortCanvas);
     scramble(size);
}
```

Das `ClockStarter`-Objekt reagiert auf einen Mausklick, indem es die Methode `startSort()` aufruft, falls der `sortThread` noch nicht erzeugt wurde bzw. schon beendet wurde. (Der `sortThread` wird in der Applet-Methode `stop()` beendet. Diese Methode wird vom Browser aufgerufen, falls die aktuelle HTML-Seite, auf der sich das Applet befindet, verlassen wird. Applets, die eigene Threads haben, sollten diese stets abbrechen oder suspendieren, wenn `stop()` aufgerufen wird.) `StartSort` erzeugt also ggf. ein neues `Thread`-Objekt, dem als `Runnable`-Parameter das Applet selbst, also `this`, übergeben wird. Beachte, daß die Methoden `start()` und `stop()` `synchronized` sind, denn sie greifen beide auf dasselbe Thread-Objekt zu.

```
     public synchronized void stop(){
          if (sortThread != null){
               sortThread.stop();
               sortThread = null;
          }
     }

     synchronized void startSort() {
          if (sortThread == null ) {
               ...
               sortThread = new Thread(this);
               sortThread.start();
          }
     }
```

```
     class ClockStarter extends MouseAdapter {
    /**
     * The user clicked in the applet.
     */
          public void mouseReleased(MouseEvent e) {
               if(sortThread == null )
                    startSort();
               else if( single_stepping )
               {
                    synchronized( SortItem.this ){
                         SortItem.this.notify();
                    }
               }
          }
    }
```

Die einzig noch fehlende Methode ist `run()`. Da das Applet das Interface `Runnable` implementiert, um als Thread laufen zu können, muß es diese Methode implementieren. Die `run()`-Methode des Applets, die durch den Aufruf von `sortThread.start()` in der Methode `startSort()` ausgeführt wird, erzeugt ein neues Objekt vom Typ `QsortAlgorithm` und initialisiert und startet es. Dieses übernimmt dann die eigentliche Sortierarbeit und teilt dem Applet nach jedem Sortierschritt mit, daß und wie es die Bildschirmdarstellung zu aktualisieren hat.

```
public void run(){
     try {
          if (algorithm == null){
              algorithm = new QSortAlgorithm();
              algorithm.setParent(this);
          }
          algorithm.init();
          algorithm.sort(arr);
     }
     catch(Exception e) {}
}
```