

Programmieren in Java

Vorlesung im Wintersemester 1997/98

Peter Pfahler

Raum: F2.311

Telefon: 606688

Email: `peter@uni-paderborn.de`

1 Objektorientierte Programmierung

Oft genannte Vorteile der OO-Programmierung:

- für **Designer**: der Entwurfsprozeß wird einfacher, klarer und handhabbarer.
- für **Programmierer**: klares Objektmodell, mächtige Programmierwerkzeuge, nützliche Bibliotheken.
- für **Manager**: Entwicklung und Wartung von Software wird schneller und billiger durch gesteigerte Produktivität.

Allerdings:

- OOP muß man lernen.
- Umso schwerer je tiefer die „prozeduralen Wurzeln“ sind.
- Gutes Design für Objekte ist nicht leicht.

Daher:

Moderne OO-Sprachen kommen mit

- Sprache
- Compiler
- Vollständige Entwicklungsumgebung
- Bibliotheken mit leicht wiederzuverwendenden Komponenten

So kann man tatsächlich mit OOP einfach und schnell zu guten Ergebnissen kommen.

1.1 Abstraktionen in Programmiersprachen

Die Komplexität der Probleme, die man lösen kann, hängt vom Grad der Abstraktion ab, den die verwendete Programmiersprache bietet:

Imperative Programmiersprachen

- Abstrahieren von der Assemblersprache
- Der Programmierer denkt eher in Begriffen der Maschine als in Begriffen des Problems.
- Zwischen Problem (*problem space*) und Maschine (*solution space*) muß eine Abbildung hergestellt werden.

Funktionale und logische Programmiersprachen

Man modelliert das zu lösende Problem.

- LISP: Alle Probleme sind letztendlich Listen.
- APL: Alle Probleme sind letztendlich mathematisch.
- PROLOG: Alle Probleme lassen sich durch Fakten und Regeln beschreiben.

Diese Ansätze sind gut für ihren jeweils typischen Anwendungsbereich. Außerhalb oft unpassend.

Objektorientierte Programmiersprachen

Der Programmierer beschäftigt sich mit Elementen aus der Problemstellung (*Objekten*) und ihrer Darstellung im Lösungsraum. Die Anwendbarkeit ist nicht auf einen bestimmten Typ von Problemen beschränkt.

„Reine“ objektorientierte Sprachen lassen sich wie folgt charakterisieren¹:

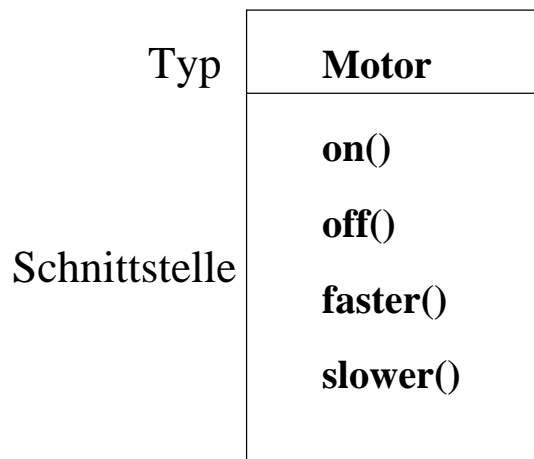
- Alles ist ein Objekt mit Eigenschaften und Verhalten.
- Ein Programm ist eine Menge von Objekten, die einander durch Botschaften mitteilen, was zu tun ist.

¹Allen Kay über Smalltalk. Vorsicht: Diese Aussagen klingen viel einfacher als sie sind.

- Jedes Objekt hat sein eigenes Gedächtnis (*Zustand*). Dieses Gedächtnis besteht aus anderen Objekten.
- Jedes Objekt hat einen Typ („ist eine Instanz einer Klasse“).
- Alle Objekte eines bestimmten Typs können die gleichen Botschaften empfangen.

1.2 Schnittstelle und Implementierung

Die Schnittstelle (*interface*) eines Objektes legt fest, welche Botschaften ein Objekt verarbeiten kann.



In Java sieht dies wie folgt aus

```
class Motor
{ public void on(){ ... }
  public void off(){ ... }
  public void faster(){ ... }
  public void slower(){ ... }
}
```

So erzeugt man einen Motor und macht ihn an:

```
Motor mo = new Motor();
mo.on();
```

- mo ist ein Behälter für einen Motor.

- `new Motor()` erzeugt ein Objekt vom Typ `Motor`.
- `=` legt diesen Motor in den Behälter `mo`.
- `mo.on()` schickt die Botschaft `on()` an diesen Motor.

Die Schnittstelle stellt die Charakterisierung eines Objektes für die Öffentlichkeit dar. Versteckt sind (i.d.R.)

- Der Zustand des Objektes, gespeichert in **Attributen**.
- Die Implementierung der Botschaften.

Wenn die Schnittstelle sich nicht ändert, kann beides ausgetauscht werden, ohne die Anwendung zu tangieren.

1.3 Wiederverwendung

Wiederverwendung und Wiederverwendbarkeit von Klassen ist eines der Schlüsselenkonzepte der objektorientierten Programmierung.

Zwei Formen:

- Komposition durch Elementobjekte
- Vererbung

1.3.1 Wiederverwendung durch Komposition

Einfachste Art der Wiederverwendung. Eine Klasse wird wiederverwendet, indem Objekte dieser Klasse als Elemente in anderen Klassen benutzt werden.

```
class Auto
{
    Motor m;
    Rad r[];
    ...
}
```

**Die Vererbung von Klassen modelliert eine
ist-ein/ist-eine-Art-von
Beziehung**

1.3.2 Wiederverwendung durch Vererbung

Vererbung schafft neue Klassen, die von ihrer Elternklasse alle Elemente und die Schnittstelle erben.

```
class Diesel extends Motor
{
    ...
}
```

Durch die Vererbung der Schnittstelle kann man an die Unterklasse die gleichen Botschaften schicken wie an die Oberklasse. Da der Typ eines Objekts definiert ist durch die Botschaften, die das Objekt versteht, folgt, daß Unter- und Oberklassenobjekte den gleichen Typ haben (*Typäquivalenz*).

Einige Sprachen definieren eine ausgezeichnete Klasse, die Oberklasse aller Klassen ist. Diese Klasse ist die Wurzel der Klassenhierarchie.

Eiffel:

GENERAL

Java, Smalltalk:

Object

Vererbung allein bringt noch nichts, da Unter- und Oberklassen nicht nur die gleiche Schnittstelle sondern auch das gleiche Verhalten haben.

Zwei Möglichkeiten:

- die Unterklasse bietet weitere Methoden an:

```
class Diesel extends Motor
{ public void glüh() { ... }
  // Nur für Diesel
}
```

- die Unterklasse ändert das Verhalten einer existierenden Basisklassenmethode durch **Überschreiben**.

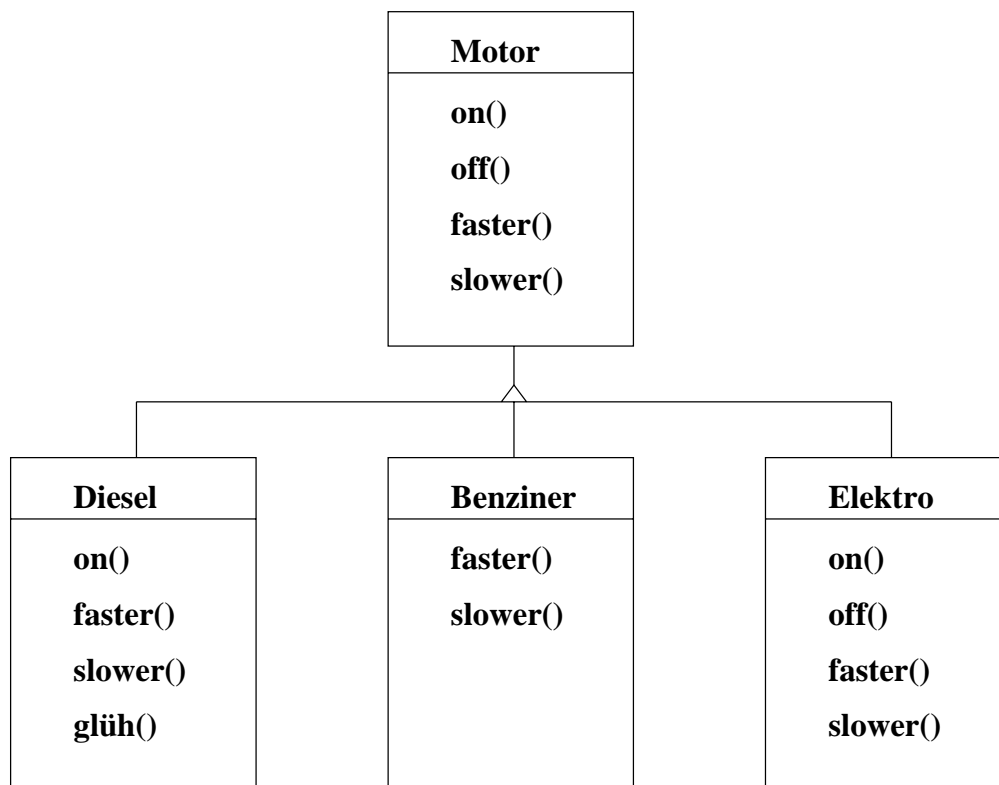
```
class Diesel extends Motor
{ public void on()
  // überschreibt "on" aus "Motor"
  { glüh();
    super.on();
  }
}
```

Je nachdem, ob die erbende Unterklasse nur Methoden überschreibt oder auch Methoden hinzufügt, haben wir zwei Ausprägungen der Beziehung:

Die Vererbung von Klassen modelliert eine
ist-ein/ist-eine-Art-von
Beziehung

1.4 Polymorphie

Vererbung führt bei einfacher Erbung (*single inheritance*) zu baumförmigen **Klassenhierarchien**.



Die wichtigste Eigenschaft von Klassenhierarchien:

**Objekte einer Unterklasse können
wie Objekte der Oberklasse
behandelt werden.**

Polymorphie heißt Vielgestaltigkeit:

```
Motor m = new Elektro();  
....  
m = new Diesel();
```

Polymorphie bedeutet:

- Wir können Code schreiben, der von spezifischen Typ-Details abstrahiert.
- Wir können Klassenhierarchien erweitern, ohne Code der Oberklassen zu invalidieren.

Beispiel:

```
void motor_check(Motor m)
{ m.on();
  // Motor-Diagnosen
  m.off();
}
...
Diesel d = new Diesel();
Elektro e = new Elektro();
motor_check(d);
motor_check(e);
```

Die Formulierung von `motor_check()` ist typunabhängig:

Nicht

```
falls Benziner: tue dies;
falls Diesel: tue das;
falls Elektro: tue jenes;
```

Sondern

```
Du bist ein Motor. Du weißt selbst,
wie du auf on() reagieren mußt.
```

`m.on()` ist der Aufruf der `on()`-Methode, die für den jeweils gültigen Typ von `m` zuständig ist (z. B. `on()` aus `Diesel`).

Welcher Typ dies ist, steht erst zur Laufzeit des Programmes fest. Der Name `on` im Aufruf `m.on()` kann erst **dynamisch** an eine der verschiedenen `on`-Ausprägungen gebunden werden.

**Polymorphie in OO Sprachen wird durch
dynamische Bindung
implementiert.**

1.5 Lebensraum und Lebensdauer von Objekten

1.5.1 Wo leben Objekte?

Statischer Speicher und Keller:

- statisch allokiert
- schnell
- unflexibel (feste Größen)
- automatisches Aufräumen

Heap:

- dynamisch allokiert
- Aufwand für Allokation und Zugriff
- flexibel (Größe wird dynamisch festgelegt)

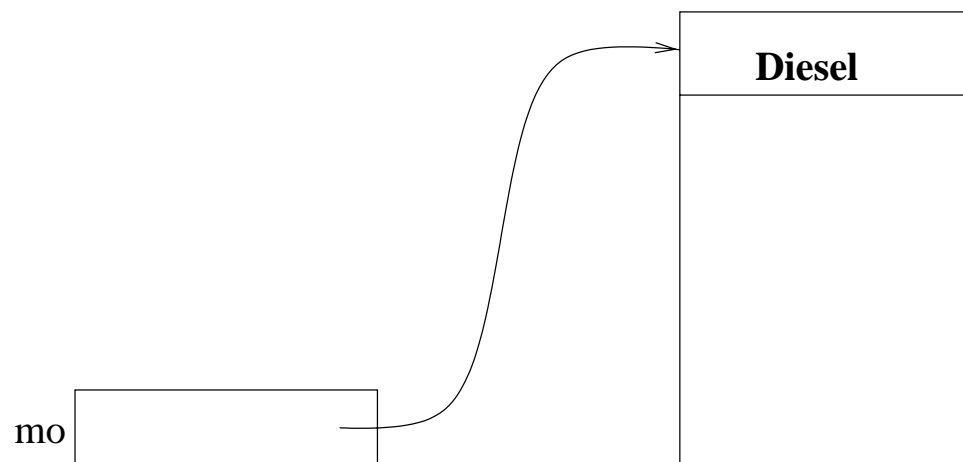
C++, Eiffel:

Freie Auswahl zwischen statischer und dynamischer Allokation

Java, Smalltalk, Delphi

Dynamischer Allokation.

Z.T. automatisches Aufräumen (*Garbage Collection*)

1.5.2 Objektzugriff über Referenzen**C++**

```
Motor *mo = new Diesel();  
...  
mo->on();
```

Referenzzugriff explizit

Java

```
Motor mo = new Diesel();  
...  
mo.on();
```

Referenzzugriff implizit

Referenzsemantik gilt für alle Typen (Objekte und Arrays)

außer für die Grundtypen (`int`, `char`, ...).

Objektkopie

Objekte werden bei Referenzsemantik immer über Referenzen manipuliert.

Bei der Zuweisung

```
a = b
```

wird also nur eine Referenz kopiert und kein Wert.

Kopie von Werten erreicht man mit

```
Vector b = new Vector();  
c = b.clone();
```

c verweist auf ein Duplikat von b.

Voraussetzung: `Vector` implementiert die Methode `clone()`.

Referenzübergabe

Objekte werden bei Referenzsemantik immer über Referenzen manipuliert.

Bei der Parameterübergabe

```
swap(a, b);
```

werden also nur Objektreferenzen übergeben.

Vorsicht: Dies hat nichts mit *Call-by-Reference* zu tun, wo Adressen von Variablen übergeben werden.

Bei Referenzsemantik der Parameterübergabe funktioniert folgende `swap`-Funktion **NICHT**:

```
public void swap(Object a, Object b)
{
    Object tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```

da sie keine Effekte auf ihre aktuellen Parameter ausübt.

1.5.3 Container und Iteratoren

Container:

Objekt, das eine *statisch unbestimmte* Zahl anderer Objekte aufnehmen kann und dafür eine bestimmte *Organisationsform* verwendet.

Beispiel:

Schlangen, Bäume, Hash-Tabellen, u.v.a.

Container sind i.a. nicht Element der Sprache sondern Teil von Bibliotheken (C++: STL, Java: API, Delphi VCL)

Iterator:

Ein Iterator für einen Container ist ein Objekt, das dem Anwender den Inhalt des Containers sequentiell präsentiert.

Beispiel:

```
Enumeration e = table.elements();
while (e.hasMoreElements())
    doSomethingWith(e.nextElement());
```

1.6 Generische Typen

Generische Typen erlauben es, Typ-Definitionen mit Typen zu parametrisieren (*Parametrische Polymorphie*).

So lassen sich z.B. Container für einen Typ von Inhalt spezialisieren:

C++

```
template <class T>
class Queue
{ ...
}
...
Queue<Person> qp;
```

Java hat keine generischen Typen.

Abhilfe: Container des allgemeinsten Typs `Object`.

```
Motor mo;    // ein Motor
Queue q;     // eine Schlange von Objects
q.enq(mo);   // mo in die Schlange
```

Nachteile:

- „wilde“ Typ-Mischung möglich
- Entnahme aus Container liefert Objekte vom Typ `Object`

Um entnommene Objekte verwenden zu können, brauchen wir Typanpassung entgegen der Typhierarchie (*Downcast*)!

```
mo = (Motor) q.deq(); // raus aus der Schlange
```

Gefahr:

Vielleicht ist `q.deq()` gar kein Motor!

Java liefert in diesem Fall eine Laufzeit-Exception. Die notwendige Prüfung kostet Laufzeit.

2 Ein Streifzug durch Java

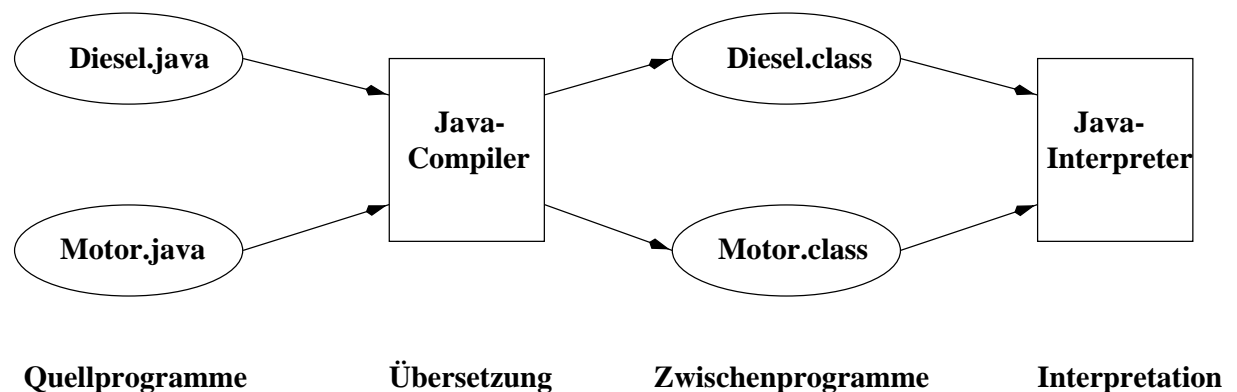
Ein erster Eindruck von Aussehen, Benutzung und Möglichkeiten von Java.

2.1 Applets und Applikationen

Zwei Einsatzbereiche:

- **Applets:** kleine meist grafikbasierte Anwendungen, die unter der Kontrolle eines Web-Browsers ablaufen.
- **Applikationen:** normale Anwendungsprogramme.

Java-Programme werden interpretiert:



Im JDK (*Java Development Kit*) von Sun sieht die Übersetzung und Ausführung **von Applikationen** so aus:

```
javac Diesel.java
javac Motor.java
java Diesel
```

falls `Diesel` das Hauptprogramm enthält.

Wichtige Regel:

Der Name der Datei (Übersetzungseinheit) ohne das “.java” muß gleich dem Namen der in der Datei definierten öffentlichen Klasse sein.

Jede Klasse in einer “.java”-Datei führt beim Übersetzen zu einer eigenen “.class”-Datei.

Dadurch:

Automatisierung des Übersetzungsprozesses:

Wenn Diesel Motor benutzt, etwa durch

```
class Diesel extends Motor
{
    ...
}
```

führt das Kommando

```
javac Diesel.java
```

automatisch auch zur Übersetzung von Motor (wenn nötig).

Applets leben in Web-Seiten. Die Applet-Zwischencode-Datei (*Bytecode*) wird im HTML-Text angegeben:

```
<html>
<head>
<title> Motor Applet </title>
</head>
<body>
<applet code=Motor.class
        width=260
        height=260>
</applet>
</body>
</html>
```

Der Interpretierer ist im Browser integriert und wird von diesem aufgerufen.

Andere Möglichkeit:

Der appletviewer aus dem JDK:

```
appletviewer slider.html
```

2.2 Hello World

```
// Datei HelloWorld.java
class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, world");
    }
}
```

! Übersetzung und Ausführung:

```
javac HelloWorld.java
java HelloWorld
```

! Definition der Klasse HelloWorld. Keine Attribute. Eine Methode.

! main ist ein spezieller Methodenname, der das Hauptprogramm kennzeichnet.

! Der Parameter ist ein Array von Strings. Es enthält die Kommandozeile.

! Die einzige Anweisung ist ein Methodaufruf. Methode: println von Objekt out aus der Klasse System.

2.3 Variablen und Ablaufstrukturen

Hier ist Java sehr C-ähnlich:

```
class Fibonacci
{
    public static void main(String[] a)
    {
        int lo = 1;
        int hi = 1;
        System.out.println(lo);
    }
}
```

```
        while (hi < 500)
        { System.out.println(hi);
          hi = lo + hi;
          lo = hi - lo;
        }
    }
}
```

! Lokale Variablen werden deklariert und evtl. initialisiert.

! Uninitialisierte Variablen gelten als *undefiniert* und führen bei Benutzung zu Compiler-Fehlermeldungen.

! Java hat 8 eingebaute Grundtypen **mit definierter** Größe.

! while-Schleife ohne Besonderheiten.

! println hier für int, d.h. println ist eine **überladene Methode**.

2.4 Kommentare

Drei Kommentar-Stile

- Zeilenkommentare: vom // bis zum Zeilenende
- Klammerkommentare: zwischen /* und */.
- Dokumentations-Kommentare: zwischen /** und */. Dokumentations-Kommentare unterstützen die automatische Erzeugung von Programmdokumentation. Ein Werkzeug namens `javadoc` extrahiert solche Kommentare und erzeugt daraus HTML Dokumentation.

Beispiel:

```
/**
 * Konstruktor mit Größenangabe
 */
schraubenschlüssel(int sz)
{ ....
}
```

2.5 Benannte Konstanten

Bekannte Vorteile gegenüber *literalen* Konstanten:

- Der Name dient der Dokumentation, wofür der Wert gut ist.
- Die Definition findet sich an einer zentralen Stelle, was die Wartung stark vereinfacht.

In Java bekommt man benannte Konstanten durch *finale Klassenvariablen*:

```
class ZylinderZahl
{ final static EINZYLINDER = 1;
  final static ZWEIZYLINDER = 2;
  final static DREIZYLINDER = 3;
}
```

`static` bedeutet: dies ist ein Attribut der Klasse (nicht von individuellen Objekten)

`final` bedeutet: dieses Attribut erhält hier seinen endgültigen Wert.

Benutzung außerhalb von `ZylinderZahl` wird qualifiziert mit dem Klassennamen:

```
if (zyl == ZylinderZahl.EINZYLINDER)
{ ....
}
```

2.6 Klassen und Objekte

Java-Klassen beinhalten zwei Arten von Elementen:

- **Attribute:** speichern den Zustand von Objekten (heißen auch *Fields*).
- **Methoden:** beinhalten den Code einer Klasse.

```
class Motor()  
{ public int rpm; // rounds per minute  
  public void on() // Motor anmachen  
  { ...  
  }  
}
```

Es gibt neben `public` noch 4 Stufen der Sichtbarkeit von Attributen und Methoden:

```
private  
protected  
public  
default: Paket-Sichtbarkeit
```

2.6.1 Erzeugen von Objekten

Objekt-Erzeugung

- immer dynamisch
- immer auf dem Heap

Zugriff immer über Referenzen (**Referenzsemantik**).

Garbage Collection (GC): automatische Entsorgung von Objekten, die nicht mehr gebraucht werden.

In Java läuft der Garbage Collector im Hintergrund immer mit.

2.6.2 Klassenvariablen = Statische Attribute

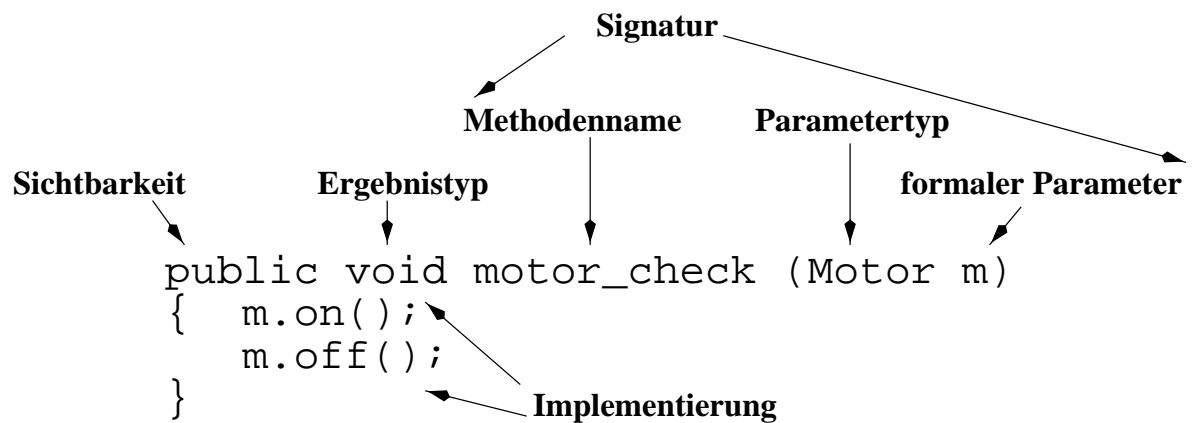
Normalerweise beschreiben Attribute die Eigenschaften von Objekten. Heißen daher auch **Instanzvariablen**.

Manchmal sollen sich alle Objekte einer Klasse auch ein Attribut teilen. Ein solches Attribut beschreibt dann Eigenschaften der Klasse.

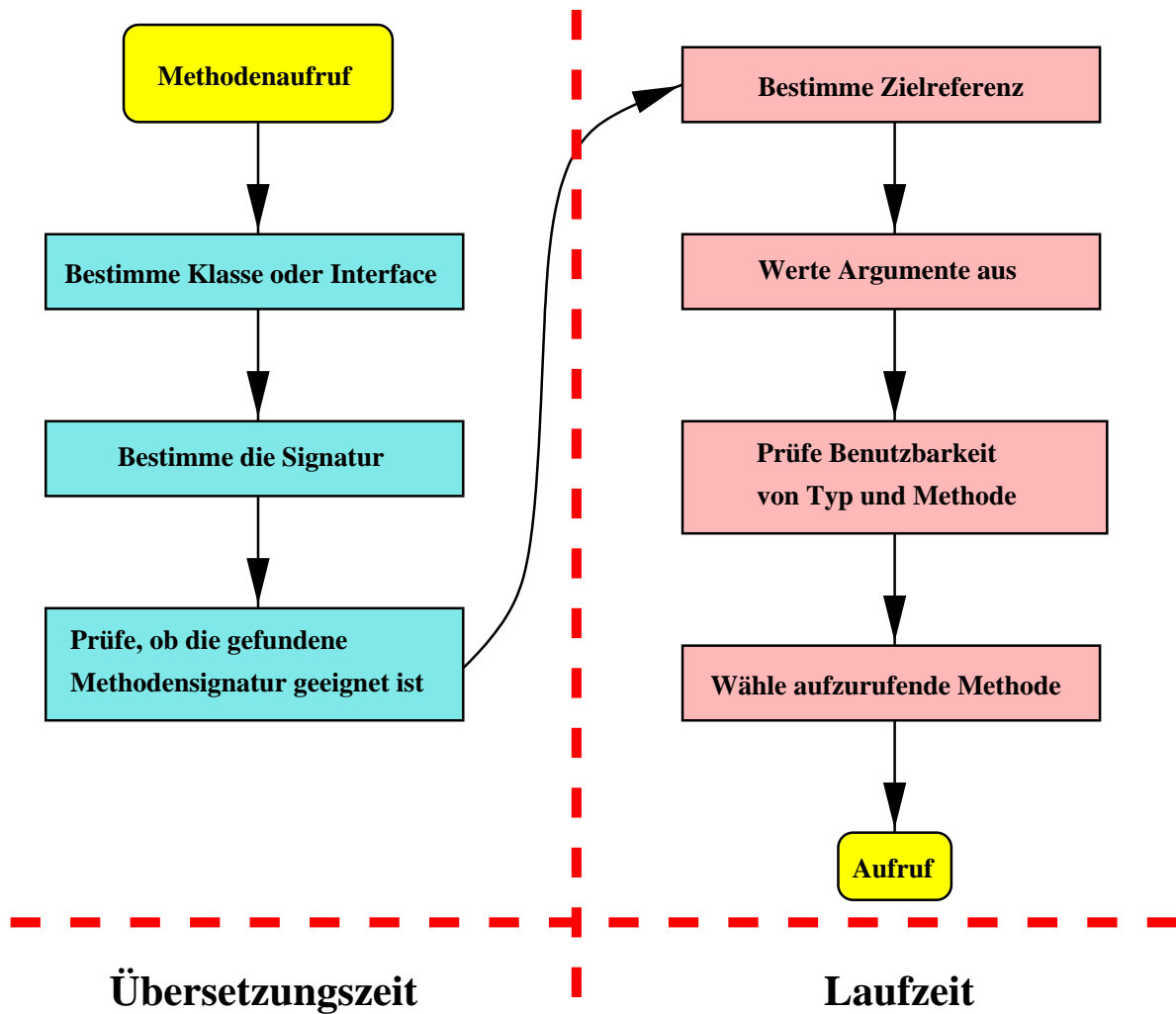
```
class WerkstattKunde
{ public static int KundenZahl;
  ...
}
```

KundenZahl ist Eigenschaft der Klasse. Alle Objekte der Klasse WerkstattKunde teilen sich dieses Attribut.

2.6.3 Methoden und Parameter



2.6.4 Methodenaufruf

2.6.5 Selbstreferenz `this`

Jede Methode erhält mit `this` eine implizite Referenz auf das Objekt, für das die Methode aufgerufen wurde.

Situation, wo man diese Selbstreferenz braucht:

```
public void on ()
{ this.glüh(); // 'glüh()' reicht auch
  super.on();
  // geht nicht ohne 'this':
  läuft.addtolist(this);
}
```

2.6.6 Klassenmethoden = Statische Methoden

Analog zu klassenweiten Attributen: **Klassenmethoden**.

Arbeiten auf Eigenschaften der Klassen, nicht auf individuellen Objekteigenschaften.

```
class WerkstattKunde
{ public static void incKundenZahl()
  { KundenZahl++;
  }
}
```

2.7 Arrays

Der zweite nicht-primitive Datentyp in Java neben Objekten.

Wie bei Objekten gilt:

- Referenzsemantik
- Dynamische Erzeugung mit new
- Automatische Entsorgung durch Garbage Collection

```
Motor Prüfstand[] = new Motor[10];
```

Jedes Array hat ein konstantes `length`-Attribut, das die Anzahl der Array-Elemente angibt:

```
for (index = 0;
    index < Prüfstand.length();
    index++)
    asu.motor_check(Prüfstand[index]);
```

In Java findet grundsätzlich **Bereichsprüfung** für Array-Zugriffe statt.

2.8 String Objekte

Strings sind eigentlich keine in Java eingebaute Klasse sondern kommen aus der Bibliothek (`java/lang/String`).

Trotzdem (!) gibt es in Java

- Literale für String-Konstanten (`"Hello, world"`)
- Automatische String-Objekt-Erzeugung für solche Literale
- Einen eingebauten Operator `+` zur Konkatenation.

```
class WerkstattKunde
{ WerkstattKunde (String name)
  { String kname = "knd_"
    + name
    + "_"
    + String.valueOf
      (KundenZahl);
    ...
  }
}
```

Wichtig: Strings sind unveränderlich (*immutable*).

Änderbare gibt es in der Klasse `StringBuffer`.

2.9 Erweitern von Klassen: Vererbung

Eine Klasse, die eine andere erweitert, erbt die Attribute und Methoden der Oberklasse.

```
class Diesel extends Motor
{ public Diesel(String n)
  { super(n); // Oberklassenkonstruktor
  }
  public void on()
  { System.out.print("Diesel ");
    glueh();
    super.on(); // Oberklassen-on()
  }
  private void glueh()
  { System.out.print("vorglühen");
  }
}
```

Möglichkeiten der Klassenerweiterung:

- Neue Attribute mit neuen Namen hinzufügen.
- Oberklassenattribute durch neue gleichnamige **verdecken**.
- Neue Methoden mit neuen Namen hinzufügen.
- Oberklassenmethoden durch neue gleichnamige mit anderer Signatur **überladen**.
- Oberklassenmethoden durch neue mit gleicher Signatur **überschreiben**.

Wörterbuch

erweitern	extend
verstecken	hide
überladen	overload
überschreiben	override

2.10 Schnittstellen (interfaces)

Reine Beschreibung des Entwurfs von Objekten. Vollständig abstrahiert von der Implementierung.

Entspricht **abstrakten Klassen** (ohne Attribute) in anderen OO-Sprachen.

Der Entwerfer von Schnittstellen spezifiziert, welche Methoden von Klassen, die die Schnittstelle **implementieren**, bereitgestellt werden.

```
interface Ersetzbar
{ // für ersetzbare Objekte lässt sich
  // eine Ersatzteilnummer ermitteln:
  int ersatzteilnummer(String Name);
}
```

Java-Klassen können mehrere Schnittstellen implementieren:

```
class Motor
implements Ersetzbar, Identifizierbar
{ ...
}
```

Schnittstellen können wie Klassen verwendet werden:

```
// Bestelle beim Großhändler alle
// benötigten Ersatzteile
public static void
  ersatzteilorder(Ersetzbar[] kaputt)
{ ...
}
```

Schnittstellen können andere Schnittstellen erweitern (auch mehrere):

```
interface DruckbarUndSpeicherbar
  extends Druckbar, Speicherbar
{ boolean aktuelleversiongedruckt();
}
```

Interfaces erweitern die Möglichkeiten der Polymorphie:

Die **Obertypen** einer Klasse T umfassen

1. den Typ, den T erweitert (*extends*)
2. die Typen, die T implementiert (*implements*)
3. die Obertypen der Typen aus [1.] und [2.].

Ein Objekt vom Typ T darf überall dort verwendet werden, wo Obertypen von T zulässig sind.

2.11 Ausnahmen (Exceptions)

Begriffe:

- *Exception*: ein Objekt, das signalisiert, daß eine außergewöhnliche Bedingung eingetreten ist.
- *throw an exception*: eine außergewöhnliche Bedingung signalisieren.
- *catch an exception*: auf die Ausnahme reagieren (wie auch immer).

Wenn eine Ausnahme nicht im Block behandelt wird, wo sie erzeugt wird, propagiert sie in den umfassenden Block, dann in die aufrufenden Methode, usw. bis zur `main`-Methode.

Wird sie auch dort nicht behandelt, erzeugt der Java-Interpreter eine Fehlermeldung und beendet die Ausführung.

Beispielprojekt: Fakultätsprogramm

Ziel:

- Fakultätsprogramm mit Argumenteingabe auf der Kommandozeile.
- Behandeln der folgenden Fehlersituationen:

- Kein Argument
- Kein ganzzahliges Argument
- Negatives Argument
- Zu großes Argument

Wir lernen:

- Ausnahmebehandlung
- Kommandozeilenverarbeitung
- Stringverarbeitung

Wir brauchen:

- Eine Klasse `fac` für die Fakultätsfunktion.
- Eine Klasse `facmain` für Kommandozeilenverarbeitung, Fehlerbehandlung und Programmsteuerung.

Die Fakultätsberechnung

```
/** Fac.java
 * Implementiert die Fakultätsfunktion
 * rekursiv.
 * Arg./Erg.: long (64bit signed integer)
 * Überprüft Fehler
 */

public class Fac
{ static final long maxarg = 20;
  public static long fac (long x)
  throws IllegalArgumentException
  {
    if (x < 0)
      throw new IllegalArgumentException
        ("negativ: " + x);
    if (x > maxarg)
```

```
        throw new IllegalArgumentException
            ("zu groß: " + x);
    if ( x == 0)
        return 1;
    else return x * fac(x-1);
}
}
```

Das Fakultäts-Hauptprogramm

```
/** facmain.java
 * Fakultätsprogramm mit
 * Kommandozeilen-Eingabe
 */
public class Facmain
{
    public static void main (String[] args)
    { // Versuche, die Fakultät zu berechnen
        try
        { int x = Integer.parseInt(args[0]);
          System.out.println(x + "! = "
                             + Fac.fac(x));
        }

        catch
        (ArrayIndexOutOfBoundsException e)
        { System.out.println("Argument fehlt!");}

        catch
        (NumberFormatException e)
        { System.out.println
          ("Argument ist keine ganze Zahl!");}

        catch
        (IllegalArgumentException e)
        { System.out.println
          ("Ungültiges Argument: "
           + e.getMessage());}
    }
}
```


2.12 Pakete (Packages)

Java benutzt Pakete zur Organisation des Namensraumes.

Ein Paket

- enthält Java-Klassen und Unter-Pakete
- hat einen Namen
- kann in Java-Programmen importiert werden

Die Paket-Hierarchie führt in voll-qualifizierter Schreibweise zu langen Typnamen:

```
java.util.Date today =  
    new java.util.Date();
```

import-Klauseln bringen Schreiberleichterung:

```
import java.util.Date; // eine Klasse  
import java.applet.*   // alles  
...  
Date today = new Date();
```

Die Zugehörigkeit zu einem Paket spezifiziert man in der 1. Anweisung einer Übersetzungseinheit:

```
package werkstatt.buchhaltung;
```

Ohne package-Anweisung gehört eine Klasse zu einem anonymen Default-Paket. Das ist praktisch für kleine Tests oder während der Entwicklung.

Java legt nicht fest, wie Pakete organisiert werden. Die meisten Java-Umgebungen bilden die Paket-Hierarchie auf die Directory-Struktur des Filesystems ab.

Java findet Klassen dann

- im aktuellen Arbeitsverzeichnis
- im dafür vorgesehenen Systemverzeichnis
- relativ zur Umgebungsvariable CLASSPATH

UNIX:

```
setenv CLASSPATH ./home/peter/werkstatt
```

WINDOWS:

```
setenv CLASSPATH .;c:\javaprogs\werkstatt
```

CLASSPATH kann statt Verzeichnissen auch zip-Dateien angeben:

```
setenv CLASSPATH ./home/jtools/cls.zip
```

3 Klassen und Objekte

Klassen sind Strukturen aus **Daten** und **Methoden**.

```
public class Kreis
{ public double x,y; // Koordinaten
  public double r;   // Radius

  public double umfang()
  { return 2*3.14159*r;
  }
  public double fläche()
  { return 3.14159*r*r;
  }
}
```

Klassen dienen (meist) dazu, den Inhalt und die Fähigkeiten von Objekten zu beschreiben.

Objekte sind **Instanzen von Klassen**.

Sie leben z.B. in Variablen von geeignetem Typ (siehe Seite 30).

```
Kreis c;
```

und werden grundsätzlich **dynamisch** erzeugt:

```
c = new Kreis();
```

Zugriff auf Objekte-Daten durch Feld-Selektion:

```
Kreis c = new Kreis();
c.x = 2.0;
```

Aufruf von Instanzmethoden durch Methoden-Selektion:

```
Kreis c = new Kreis();  
double f;  
c.r = 2.7;  
f = c.fläche();
```

Technische Realisierung: Übergabe der Selbstreferenz `this` (siehe Seite 25) an den Code, der `fläche` implementiert.

3.1 Konstruktoren

```
Kreis c = new Kreis();
```

Ablauf:

- dynamische Allokation des neuen Objektes
- Aufruf des Konstruktors `Kreis()`

Konstruktoren initialisieren Objekte.

Wichtige Eigenschaften:

- Sie tragen den Namen der Klasse.
- Sie haben keinen Ergebnistyp.
- Sie geben implizit das `this`-Objekt zurück, haben also keine `return`-Anweisung.
- Sie sind überladen, d.h. es kann mehrere Konstruktoren mit unterschiedlicher Signatur geben.

```
public Kreis (double x, double y,  
              double r)  
{ this.x = x; // 'this' löst  
  this.y = y; // Namens-  
  this.r = r; // konflikt  
}  
...  
Kreis c = new Kreis(1.4,2.0.5.3);
```

Der parameterlose Konstruktor

```
Kreis();
```

heißt **Default-Konstruktor**. Er wird vom Compiler zur Verfügung gestellt, falls es keinen Benutzer-definierten Konstruktor gibt.

Der Compiler-generierte Default-Konstruktor ruft lediglich den Default-Konstruktor der Oberklasse auf.

Wir bekommen also Compiler-Fehlermeldungen,

- wenn der Compiler-generierte Default-Konstruktor aufgerufen wird und die Oberklassen keinen Default-Konstruktor hat.
- wenn ein Default-Konstruktor benutzt wird für Klassen, die Konstruktoren mit Parametern haben.

3.1.1 Die Selbstreferenz `this` in Konstruktoren

`this` findet in Konstruktoren Anwendung, die andere Konstruktoren verwenden:

```
public Kreis (double x, double y,
              double r)
{ this.x = x;
  this.y = y;
  this.r = r;
}
public Kreis(double r)
{ this(0.0, 0.0, r);
}
public Kreis(Kreis c)
{ this(c.x, c.y, c.r);
}
public Kreis()
{ this(0.0, 0.0, 1.0);
}
```

Andere typische Anwendungen von `this` siehe Seite 25 und Seite 36.

Solche Aufrufe anderer Konstruktoren stehen **vor** anderen Anweisungen im Konstruktorrumpf.

Solch ein expliziter Konstruktor-Aufruf über `this`

- darf nicht rekursiv sein.
- darf in seinen Ausdrücken keine Instanzvariablen oder Methoden verwenden.
- darf in seinen Ausdrücken nicht `this` oder `super` benutzen.

All dieses führt zu Compiler-Fehlermeldungen.

```
public class Tisch
{ Bein[] beine;
  int anz_beine;
  public Tisch()
  { this(anz_beine-1); // FALSCH!
  }
  public Tisch(int n)
  { anz_beine = n;
  }
}
```

Der Java-Compiler meldet:

```
Can't reference anz_beine before the
superclass constructor has been called.
{ this(anz_beine-1);
  ^
```

3.2 Klassenvariablen

Klassenvariablen beschreiben im Gegensatz zu Instanzvariablen Eigenschaften der Klasse (siehe auch Seite 24).

Klassenvariablen werden mit dem Schlüsselwort `static` deklariert:

```
public class Kreis
{
    static int anz_Kreise = 0;
    public double x,y; // Koordinaten
    public double r;    // Radius
    public Kreis (double x, double y,
                  double r)
    {
        this.x = x; this.y = y; this.r = r;
        anz_Kreise++;
    }
}
```

Zugriff erfolgt außerhalb der Klassen qualifiziert mit dem Klassennamen:

```
if (Kreis.anz_kreise == 12)
    System.out.println(`Hello`);
```

static-Felder werden außerdem benutzt für

- Ersatz für globale Variablen (`public static`)
- Konstanten (`public static final`)

(Konstanten siehe Seite 22)

3.3 Klassenmethoden

analog zu den Klassenvariablen:

- werden als `static` deklariert
- werden mit dem Klassennamen qualifiziert

```
public class Kreis
{
    public double x,y; // Koordinaten
    public double r;    // Radius

    // Instanz-Methode:
```

```
public Kreis bigger(Kreis k)
{ if (k.r > r)
    return k;
  else return this;
}

// Klassen-Methode:
public static Kreis bigger(Kreis k1,
                           Kreis k2)
{ if (k1.r > k2.r)
    return k1;
  else return k2;
}
```

Verwendung der beiden Methoden:

```
Kreis a = new Kreis(12.3);
Kreis b = new Kreis(3.414);
Kreis c = a.bigger(b);
Kreis c = Kreis.bigger(a,b);
```

Besonderheiten:

- Es gibt (natürlich) keine implizite Selbstreferenz `this`.
- Statische Methoden dürfen nur auf Klassenvariablen ihrer Klasse zugreifen, nicht auf Instanzvariablen.
- Statische Methoden dürfen nur Klassenmethoden ihrer Klasse aufrufen, keine Instanzmethoden.

```
// Klassen-Methode:
public static Kreis bigger(Kreis k1,
                           Kreis k2)
{ if (k1.r > k2.r &&
```



```
        x > 0.0)          // FALSCH!  
        return k1;  
    else return k2;  
}
```

Java-Compiler:

```
Can't make a static reference to  
nonstatic variable x in class Kreis.  
{ if (k1.r > k2.r && x > 0.0)  
    ^
```

3.4 Initialisierer

Attribute können bei der Deklaration initialisiert werden. Dies gilt sowohl für

Instanzvariablen

als auch für

Klassenvariablen

```
class Kreis  
{ double r = 1.0;  
  static int kreiszahl = 100;  
  ...  
}
```

Instanzvariablen:

Initialisierung wird jedesmal bei Erzeugung eines Objektes ausgeführt.

Klassenvariablen:

Initialisierung wird nur einmal durchgeführt, wenn die Klasse geladen wird.

3.4.1 Initialisierung von Klassenvariablen

```
class Rechteck
{ static float max_sv = 2.0;
  static float max_h = 10.0
  static float max_b = max_sv * max_h;
  ...
}
```

Die Regeln:

- Der Initialisierungsausdruck darf nicht rekursiv sein.
- Der Initialisierungsausdruck darf keine Vorwärtsreferenz auf andere Klassenvariablen enthalten.
- Der Initialisierungsausdruck darf keine Instanzvariablen enthalten.
- Der Initialisierungsausdruck darf nicht `this` oder `super` verwenden

Daher bekommen wir hiermit 2 Compiler-Fehlermeldungen:

```
class FalscheStaticInits
{ static float f = j;          // VORWÄRTS!
  static int j = 1;
  public int max;
  static int k = max + 1; // INSTANZVAR!
  ...
}
```

Für komplexere Initialisierungen, die nicht als einfacher Ausdruck geschrieben werden können, gibt es die **statischen Initialisierungsblöcke**:

```
public class Kreis
{ static private double pi = 3.14;
  static private double[] sins
    = new double[1000];

  // Statischer Initialisierungsblock
  static
```

```

    { double x = 0.0;
      double delta_x = pi / 2 / (1000-1);
      int i;
      for (i=0; i < 1000; i++)
        { sins[i] = Math.sin(x);
          x += delta_x;
        }
    }
}

```

Es darf mehrere solcher Blöcke geben. Sie werden behandelt wie ein einziger, der durch Hintereinanderhängung entsteht.

3.4.2 Initialisierung von Instanzvariablen

```

public class Tisch
{ Bein[] beine;
  int anz_beine = 4;
}

```

Hier gelten ähnliche Regeln wie bei den Klassenvariablen (Seite 42):

- Der Initialisierungsausdruck darf nicht rekursiv sein.
- Der Initialisierungsausdruck darf keine Vorwärtsreferenz enthalten.
- Der Initialisierungsausdruck darf **alle** Klassenvariablen der Klasse benutzen.
- Der Initialisierungsausdruck darf `this` und `super` benutzen.

Daher bekommen wir hiermit 2 Compiler-Fehlermeldungen:

```

class FalscheInstanzInits
{ float f = j;          // VORWÄRTS!
  int j = 1;
  int k = k + 1;        // REKURSIV!
  ...
}

```

Für komplexerer Initialisierung, die nicht als einfacher Ausdruck geschrieben werden können, gibt es die **Instanz-Initialisierungsblöcke** (ab Java1.1):

```
public class Tisch
{ String platte = "Holz";
  String[] beine;
  int anz_beine = 4;

  { int i;
    for (i = 0; i < anz_beine; i++)
      beine[i] = "Bein " + i;
  }

  public Tisch(String p_material)
  { platte = p_material;
  }
}
```

Es darf mehrere solcher Blöcke geben. Sie werden behandelt wie ein einziger, der durch Hintereinanderhängung entsteht.

Nun kennen wir alle Akteure, die bei der Erzeugung neuer Objekte eine Rolle spielen:

- Initialwerte
- Instanz-Initialisierungsblöcke
- Konstruktoren
- Oberklassen-Konstruktoren

3.5 Objekterzeugung

Wann werden neue Objekte erzeugt?

- Bei der Auswertung von “*Klasseninstanzerzeugungsaustrücken*”:

```
new Tisch();
new Kreis(3.2);
```

- bei Aufruf der `newInstance()`-Methode der Klasse `Class` (aus `java.lang.Class`, siehe Seite 45).
- Laden einer Klasse, die ein String-Literal enthält: dieses wird in ein String-Objekt umgewandelt.
- Wenn nötig: Für Zwischenergebnisse (String-Objekte) bei der Auswertung des String-+ Operators. .

Einschub: Die Klasse `Class`

Für

- Jede verwendete Klasse
- Jedes verwendete Interface
- Jedes verwendete Array bestimmten Typs und bestimmter Dimensionsanzahl

existiert zur Laufzeit eines Java-Programms ein Objekt der Klasse `Class`

Diese Objekte können nur von der virtuellen Maschine erzeugt werden, nicht vom Benutzer-Programm.

Zu jedem Objekt findet man das Klassenobjekt durch `getClass()` aus der Klasse `Object`:

```
Class c = new Diesel().getClass();
```

Man kann nun z.B. den Klassennamen ausgeben:

```
Class c = new Diesel().getClass();  
System.out.println(c.getName());
```

Nachzulesen in der API-Dokumentation zu `java.lang.Class`.

[Ende des Einschubs.]

Der Ablauf der Objekterzeugung:

1. Speicherallokation
2. Default-Initialisierung
- 3.. Konstruktion des Objektes

3.5.1 Speicherallokation

Reserviere einen Speicherbereich, groß genug für

- alle Instanzvariablen der Klasse des Objektes
- und alle Instanzvariablen aller Oberklassen des Objektes (auch die versteckten)

Gelingt dies nicht, gibt es einen `OutOfMemoryError`-Fehler.

Gelingt es, werden anschließend alle Instanzvariablen (auch die der Oberklassen) des neuen Objektes mit ihren Default-Werten initialisiert.

3.5.2 Default-Initialisierung

Typ	Inhalt	Größe	Default
<code>boolean</code>	<code>true</code> oder <code>false</code>	1 bit	<code>false</code>
<code>char</code>	Unicode Character	16 bits	<code>\u0000</code>
<code>byte</code>	signed integer	8 bits	0
<code>short</code>	signed integer	16 bits	0
<code>int</code>	signed integer	32 bits	0
<code>long</code>	signed integer	64 bits	0
<code>float</code>	IEEE 754 floating point	32 bits	0.0
<code>double</code>	IEEE 754 floating point	64 bits	0.0
Klasse, Interface, Array	Referenz	/	<code>null</code>

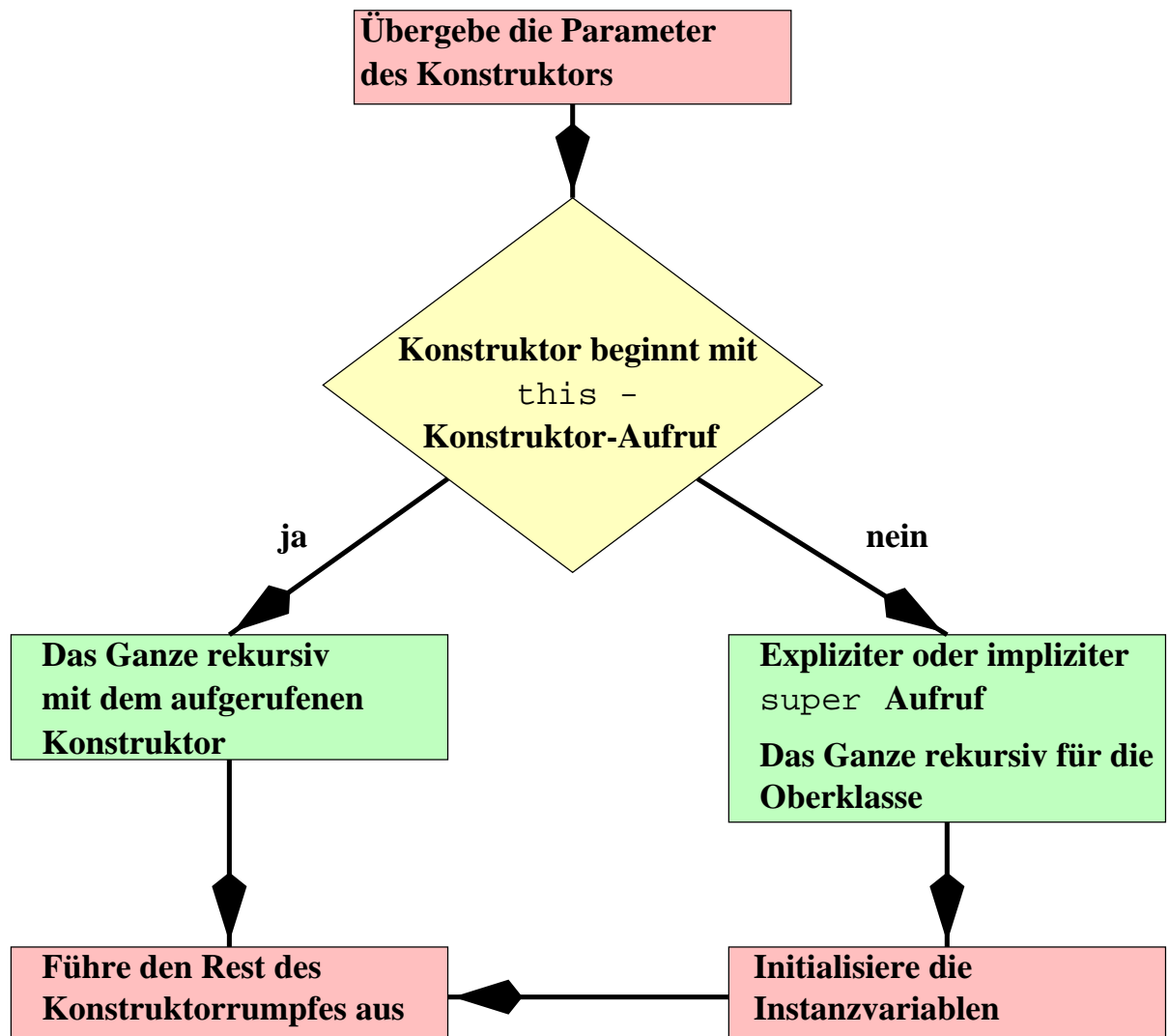
`null` ist ein Ausdruck von einem speziellen Nulltyp, der keinen Namen hat.

`null` kann an jede Array-, Objekt- oder Interface-Variable zugewiesen werden.

Die Größe von Referenzen wird von der virtuellen Maschine als ein “Wort” definiert (i.d.R. 32 oder 64 bits).

3.5.3 Konstruktion von Objekten

Ablauf:



3.6 Objektzerstörung

Nicht unter der Kontrolle des Benutzerprogramms sondern durch einen automatischen **Garbage Collector**.

Der weiß zur Laufzeit, welche Objekte existieren und prüft, auf welche Objekte nicht mehr verwiesen wird.

Technisch: Low-priority-thread, der im Hintergrund mitläuft. Nutzt z.B. die Zeiten des Wartens auf Benutzereingabe.

Hohe Priorität bekommt der Garbage Collector nur, wenn der Interpreter keinen Speicher mehr hat.

Für die Entsorgung von Objekten muß also nichts getan werden.

Ein Beispielfall, wo wir dem Garbage Collector helfen können:

```
public static void meth ()
{ int big_array[] = new int[100000];
  // rechne auf big_array
  int result = compute(big_array);

  // brauchen big_array nicht mehr
  // nach verlassen von meth kann es
  // entsorgt werden.
  // falls meth aber NICHT verlassen
  // wird:

  big_array = null;

  for (;;)    // für immer
    handle_user_input();
}
```

3.6.1 Finalisierung von Objekten

Um die Freigabe der Ressource “Speicher” kümmert sich der Garbage Collector. Andere Ressourcen können in der **finalizer**-Methode freigegeben werden.

Hier ist die finalizer-Methode aus

java.io.FileOutputStream:


```
public void finalize()  
    throws IOException;  
    // overrides Object  
{ // close the stream  
    if (fd != null)  
        close();  
}
```

Regeln für Finalizer:

- Die `finalize`-Methode wird vor der Garbage-Collection aufgerufen.
- Java gibt keine Garantie, ob und wann Objekte entsorgt werden. Daher auch keine Garantie für die Ausführung von `finalize`-Methoden.
- Nach der Finalisierung folgt nicht unmittelbar die Entsorgung.
Grund: `finalize`-Methoden können Objekte “wiederauferstehen” lassen (z.B. durch Speichern des `this`-Zeigers).²
GC muß also erneut prüfen.
- `finalize`-Methoden werden höchstens einmal aufgerufen!

3.7 Innere Klassen**In Java 1.0:**

- Klassen und Interfaces nur “top-level” als Teile von Paketen.

In Java 1.1 zusätzlich:

- Geschachtelte “top-level” Klassen und Interfaces – in andere Klassen oder Interfaces geschachtelt, trotzdem “top-level”.
- Elementklassen – innere Klassen in anderen Klassen.
- Lokale Klassen – lokal in einem Java-Block.

²Niemand kennt eine sinnvolle Anwendung von “Objekt-Wiederauferstehung”.

- Anonyme Klassen – Namenlose Klasse, von der es nur eine Instanz gibt.

Ziel: Definition von Hilfsklassen möglichst nahe an der Stelle, wo sie gebraucht werden.

Geschachtelte und innere Klassen sind wesentlich motiviert durch das neue Ereignismodell im AWT von Java1.1.

3.7.1 Geschachtelte “top-level” Klassen und Interfaces

Klassen oder Interfaces, die `static` deklariert sind und in andere “top-level”-Klassen oder Interfaces eingeschachtelt sind.

Interfaces sind immer `static`, d.h. für Interfaces ist nur die “top-level”-Variante möglich.

Geschachtelte “top-level” Klassen und Interfaces verhalten sich wie andere Paket-Elemente auf der äußersten Ebene, außer daß ihrem Namen der Name der umgebenden Klasse vorangestellt wird.

```
public class LinkedList
{ // interface verkettungselement
  public static interface Linkable
  { public Linkable getNext();
    public void setNext(Linkable n);
  }
  // kopf der liste:
  Linkable head;
  // einfügen:
  public void insert(Linkable n)
  { n.setNext(head);
    head = n;
  }
  // kopfzugriff:
  public Linkable getHead()
  { return head;
  }
}
```

Benutzung:

a) Definition der Verkettungselemente:

```
class LinkableInteger
implements LinkedList.Linkable
{
    int i;
    LinkedList.Linkable Next;
    // konstruktor:
    LinkableInteger(int i)
    { this.i = i;
    }
    // versprochene Methoden:
    public LinkedList.Linkable getNext()
    { return Next;
    }
    public void setNext(LinkedList.Linkable n)
    { Next = n;
    }
    // für die ausgabe: String-Umwandlung
    // überschreibe toString aus Object
    public String toString()
    { return i + "";
    }
}
```

b) Testlauf:

```
public class LiMain
{
    public static void main (String[] argv)
    {
        LinkedList li = new LinkedList();
        li.insert(new LinkableInteger(16));
        li.insert(new LinkableInteger(6));
        li.insert(new LinkableInteger(-4));

        LinkedList.Linkable l;
        for (l = li.getHead();
            l != null;
            l = l.getNext())
```

```
        System.out.println(1);  
    }  
}
```

c) Ergebnis:

```
-4  
6  
16
```

3.7.2 Elementklassen

Im Gegensatz zu den `static`-deklarierten geschachtelten “top-level”-Klassen, die mehr oder weniger nur zur Strukturierung dienen, sind Elementklassen echte **innere Klassen**.

**Jeder Instanz einer inneren Klasse ist ein
Objekt der umfassenden Klasse
zugeordnet.**

Damit kann das Objekt der Elementklasse implizit auf die Instanzvariablen der umgebenden Klasse zugreifen (auch auf `private`).

Die umgebende Klasse hat Zugriffsrechte auf alle Felder aller ihrer Elementklassen. Zwei Elementklassen der selben umgebenden Klasse haben Zugriffsrechte auf die Elemente des jeweilig anderen.

Elementklassen dürfen keine statischen Elemente (Attribute, Methoden, Klassen, Interfaces) besitzen. Gilt auch für lokale Klassen (Seite 62) und anonyme Klassen (Seite 63).

Beispielprojekt: Einbau eines Iterators in eine Buchstabenmenge

Beschreibung der Buchstabenmenge:

Name	Charset
Attribut	chars vom Typ BitSet
Konstruktor	Charset(String)
Methode	toString() schreibt die Elemente in einen String.

Ziel: ein Iterator für Charset.

Typisches Verwendungsmuster für einen solchen Iterator:

```
Charset cs = new Charset(somestring);
Enumeration e = cs.characters();
while (e.hasMoreElements())
    machwasmit(e.nextElement());
```

(Anderes Iteratorbeispiel auf Seite 16).

a) Originalversion der Buchstabenmenge:

```
import java.util.BitSet;

class Charset
{ // Buchstabenmenge:
  private BitSet chars = new BitSet();
  // Konstruktor:
  public Charset(String str)
  { for (int i = 0; i < str.length(); i++)
```

```

        chars.set(str.charAt(i));
    }
    // Umwandlung in String:
    public String toString()
    { String result="[";
      int size = chars.size();
      for (int i = 0; i < size; i++)
          if (chars.get(i))
              result += (char)i;
      return result +"]";
    }
}

```

b) Testen der Originalversion der Buchstabenmenge:

```

public class CharsetMain
{ // hauptprogramm
  public static void main(String argv[])
  { // erstes kommandozeilenargument
    Charset cs = new Charset(argv[0]);
    System.out.println(cs + "");
  }
}

```

c) Testlauf:

Eingabe:

```
java CharsetMain JavaProgrammierung
```

Ausgabe:

```
[JPaegimnoruv]
```

c) Einbau einer inneren Iteratorklasse

```
import java.util.*
// für BitSet, Enumeration
// und NoSuchElementException

class CharSet
{ // Buchstabenmenge:
  private BitSet chars = new BitSet();

  // Konstruktor:
  public CharSet(String str)
  { for (int i = 0; i < str.length(); i++)
    chars.set(str.charAt(i));
  }

  // innere Elementklasse:
  private class Cs_iterator
  implements Enumeration
  { private int pos;
    private int setsize = chars.size();

    // versprochenes "hasMoreElements":
    public boolean hasMoreElements()
    { while (pos < setsize &&
      !chars.get(pos))
      pos++;
    return pos < setsize;
  }

  // versprochenes "nextElement":
  public Object nextElement()
  throws NoSuchElementException
  { if (hasMoreElements())
    return new Character((char) pos++);
    else
    throw new NoSuchElementException();
  }
}

// liefert einen Iterator:
```

```
    public Enumeration characters()  
    { return new cs_iterator();  
    }  
}
```

d) Testen des neuen Charset mit Iterator:

```
import java.util.*;  
  
public class It_CharsetMain  
{ public static void main(String argv[])  
  { // erstes kommandozeilenargument  
    Charset cs = new Charset(argv[0]);  
  
    Enumeration e = cs.characters();  
    while (e.hasMoreElements())  
      System.out.println(e.nextElement());  
  }  
}
```

e) Testlauf:

Eingabe:

```
java It_CharsetMain Java
```

Ausgabe:

```
J  
a  
v
```

Syntax-Erweiterungen und neue Regeln für Elementklassen**a) Wie sprechen “innere Objekte” das ihnen zugeordnete Objekt der umgebenden Klasse an?**

Im Charset-Beispiel:

```
while (pos < setsize && !chars.get(pos))
```

Instanzvariable der inneren Klasse

Instanzvariable der umgebenden Klasse

`pos` und `setsize` sind Attribute der Selbstreferenz `this`.

Ihr expliziter Name wäre also `this.pos`, bzw. `this.setsize`.

Um `chars` explizit anzusprechen, schreiben wir:

```
Charset.this.chars
```

b) Tiefere Schachtelung von Elementklassen?

Beliebig möglich. Allerdings darf keine der eingeschachtelten Klassen so heißen wie eine der sie umgebenden!

Das bedeutet:

Obige neue Syntax:

```
classname.this
```

ist völlig ausreichend:

```
public class A
{ public string name = "a";
  public class B
  { public string name = "b";
```

```

    public class C
    { public string name = "c";
      public void print()
      { System.out.println(name);
        System.out.println(this.name);
        System.out.println(C.this.name);
        System.out.println(B.this.name);
        System.out.println(A.this.name);
      }
    }
  }
}

```

c) Welches ist das umgebende Objekt?

In unserem Beispiel haben wir den Default-Fall benutzt:

```

public Enumeration characters()
{ return new cs_iterator();
}

```

Hier: Keine Angabe des umgebenden Objektes für die neue Instanz der inneren Klasse. Daher wird die Selbstreferenz `this` als umgebendes Objekt angenommen:

Äquivalent zu obigem ist die neue **explizite Schreibweise**:

```

public Enumeration characters()
{ return this.new cs_iterator();
}

```

Für andere Fälle brauchen wir die neue **allgemeine Syntax**:

```

containing_instance.new inner_class(...)

```

wobei `containing_instance` eine Instanz der umfassenden Klasse von `inner_class` ist.

d) Innere Objekte außerhalb der umgebenden Klasse erzeugen?

Geht, wenn die innere Klasse nicht wie in unserem Charset-Beispiel privat ist.

Also ändern:

```
// innere Elementklasse:
public class Cs_iterator
implements Enumeration
{ private int pos;
  ....
}
```

Dann funktioniert auch folgendes Hauptprogramm:

```
public static void main(String argv[])
{ // erstes kommandozeilenargument
  Charset cs = new Charset(argv[0]);

  Enumeration e = cs.new cs_iterator();
  while (e.hasMoreElements())
    System.out.println(e.nextElement());
}
```

Noch ein Beispiel für die externe Erzeugung innerer Objekte. Klassenhierarchie von Seite 58:

```
A a = new A;           // A Instanz
A.B b = a.new B();     // B Instanz in a
A.B.C c = b.new C();   // C Instanz in b
c.print();             // Methode aus C
```

e) Kann man von inneren Klassen erben?

Ja (*“strange as it may seem”*).

Damit gibt es aber auch eine zweite Art, wie Objekte innerer Klassen erzeugt werden können: mit dem Oberklassen-Konstruktoraufruf `super`.

Zusätzlich zur Syntax

```
containing_instance.new inner_class(...)
```

brauchen wir also

```
containing_instance.super(...)
```

f) Ändern sich durch innere Klassen die Regeln für die Namensbindung?

Ja. Wir haben jetzt **drei** Hierarchien in Java-Programmen:

1. die klassische Blockschachtelung im Anweisungsteil
2. die Vererbungshierarchie
3. die Klassenschachtelung

Aufgabe der Namensanalyse: Finde zu jedem Auftreten eines Bezeichners die laut Sprachdefinition dazugehörige Definition.

Beispiel:

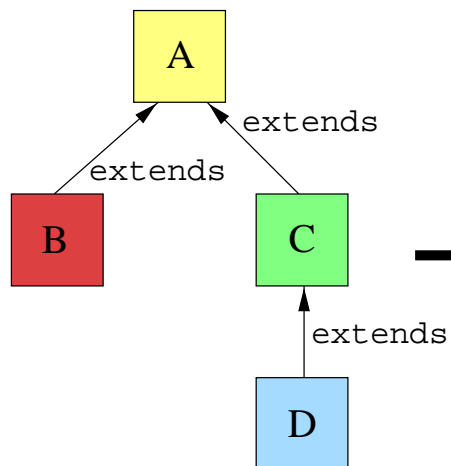
Eine Methode der Klasse `K` verwendet in einer Anweisung den Bezeichner `x`.

Ist `x`

- Die lokale `double`-Variable `x` dieser Methode ?
- Die `String[]`-Instanzvariable `x` der Klasse `K` ?
- Die `Date`-Instanzvariable `x` der Oberklasse von `K` ?

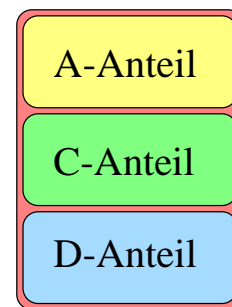
- Die `int`-Instanzvariable `x` im umfassenden Objekt ?
- ?

Vererbungshierarchie und Klassenschachtelung im Bild:

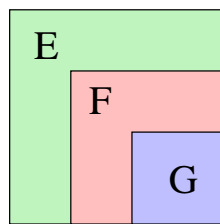


Klassenhierarchie

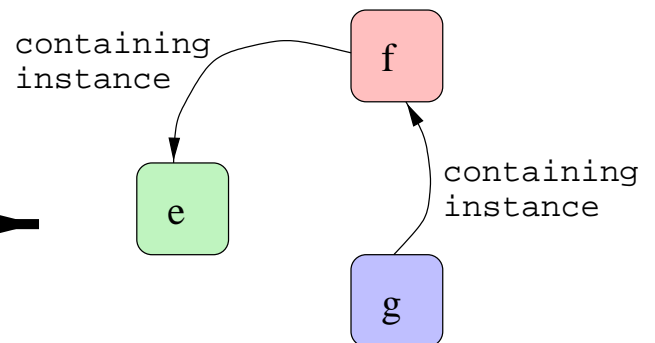
```
D d = new D();
```



Objektstruktur



Klassenschachtel-Struktur



Objektschachtel-Struktur

Die beiden Hierarchien sind völlig unabhängig voneinander.

Es ist wichtig, sie nicht zu verwechseln!

Konfliktfall zwischen umfassender Klasse und Oberklasse

In Java 1.1. gelöst zugunsten der Oberklasse, d.h.

der Bezeichner `x` der Oberklasse verdeckt den der umfassenden Klasse.

Zusätzlich zu dieser Regel erfordert Java im Konfliktfall die **explizite** Benennung des Bezeichners, d.h.

- `this.x` für das geerbte Feld, bzw. `super.x`
- `A.this.x` für das Feld der umfassenden Klasse `A`.

3.7.3 Lokale Klassen

Lokale Klassen sind innere Klassen, die nicht auf oberer Ebene in andere Klassen eingeschachtelt sind, sondern lokal in Anweisungsblöcken von Methoden, statischen Initialisierern oder Instanz-Initialisierern.

Sehr ähnlich zu Elementklassen (siehe Seite 52) mit folgenden wichtigen Unterschieden:

- Nur in dem Block, der sie definiert, sind lokale Klassen benutzbar. Hilfsklassen können also noch näher an ihre Anwender heran.
- Die Modifizierer `public`, `protected`, `private` und `static` sind **nicht** vor dem Klassennamen erlaubt.
- In lokalen Klassen können alle sichtbaren lokalen Variablen und Parameter benutzt werden, die als **final** deklariert sind (Finaldeklaration für Variablen und Parameter neu in Java 1.1).

3.7.4 Anonyme Klassen

Anonyme Klassen werden wie lokale Klassen (siehe Seite 62) z.B. innerhalb von Anweisungsblöcken definiert.

Anonyme Klassen haben keinen Namen. Sie entstehen immer zusammen mit einem Objekt.

Ihr Zweck: “Einweg-Klassen”

```
import java.io.*;
// gibt die Namen aller .java-Dateien im
// angegebenen Verzeichnis aus
public class Dirlist
{public static void main(String argv[])
{ File f = new File(argv[0]);
  String[] list =
    f.list (new FilenameFilter()
    { public boolean accept(File f,
                          String s)
      { return s.endsWith(".java");
      }
    }
    ); // Formatierung schwierig!
  for (int i = 0; i < list.length; i++)
    System.out.println(list[i]);
}
}
```

Die Syntax für anonyme Klassen ist

```
new-expression class-body
```

Für die anonyme Klasse kann man keine `extends`- oder `implements`-Klauseln angeben.

Daher:

- Wenn hinter `new` ein Klassennamen folgt, erbt die anonyme Klasse von dieser.
- Wenn hinter `new` ein Interface-Namen folgt, implementiert die anonyme Klasse dieses und erbt von `Object`.

In unserem Beispiel implementiert die anonyme Klasse ein Interface:

```
f.list(new FilenameFilter()
```

```

    { ...
    }
):

```

Da anonyme Klassen keinen Namen haben, können sie auch keine Konstruktoren haben. Deshalb hat Java 1.1 die Instanzinitialisierungsblöcke eingeführt (Seite 44).

3.8 Sichtbarkeitsspezifikationen

Die Sichtbarkeitsspezifikationen

- `public`
- `protected`
- `private`
- Default-Sichtbarkeit "Paket"

geben an, wo Attribute und Methoden einer Klasse benutzt werden können.

Benutzbar in	Sichtbarkeit			
	<i>public</i>	<i>protected</i>	<i>Paket</i>	<i>private</i>
Selber Klasse	ja	ja	ja	ja
Klasse, selbes Paket	ja	ja	ja	nein
Unterklasse, anderes Paket	ja	ja	nein	nein
Klasse, anderes Paket	ja	nein	nein	nein

Besonderheit im Zugriff auf `protected`-Elemente aus Unterklassen:

Erlaubt ist der Zugriff auf `protected`-Elemente des eigenen Oberklassenanteils.

Nicht Erlaubt ist der Zugriff auf `protected`-Elemente in anderen Objekten vom Oberklassentyp.

Beispiel:

Klasse A im Paket pack:

```
package pack;

public class A
{ protected int x;
}
```

Klasse B in einem anderen Paket:

```
class B extends pack.A
{ void doit()
  { x = 18;
    pack.A va = new pack.A();
    va.x = 19;    // NICHT ZULÄSSIG !!
  }
}
```

4 Vererbung

Neben der Komposition (Seite 7) die wichtigste Form der Wiederverwendung auf der Basis von Klassen.

Eine Oberklasse

```
class Point
{ private float x,y;
  static int pcount = 0;

  public Point(float x,float y)
  { this.x = x;
    this.y = y;
    pcount++;
  }

  public Point()
  { this(0.0, 0.0);
  }

  public void move(float dx, float dy)
  { x += dx;
    y += dy;
  }
}
```

wird zu Implementierung einer neuen Klasse wiederverwendet:

```
class Point3D extends Point
{ private float z;

  public Point3D(float x, float y, float z)
  { super(x, y);
    this.z = z;
  }
}
```

```
public Point3D()  
{ this(0.0, 0.0, 0.0);  
}  
  
public void move(float dx,  
                 float dy,  
                 float dz)  
{ move(dx, dy);  
  z += dz;  
}  
}
```

Jede Java-Klasse benutzt Vererbung:

- **explizit** spezifiziert durch `extends`
- oder **implizit**: Klasse `Object` ist Oberklasse.

Durch direkte oder indirekte Vererbung ist `Object` die Oberklasse aller Java-Klassen.

Wie sieht `Object` aus?

Information aus der Dokumentation der JDK APIs.

Oder direkt von `java.sun.com`

Constructor Index

```
public Object();
```

Method Index

```
public final native Class getClass()
```

```
public native int hashCode()  
public boolean equals(Object obj)  
public native Object clone()  
public String toString()  
protected void finalize()
```

außerdem 5 Methoden zur thread-Behandlung:

```
notify, notifyAll, wait
```

4.1 Vererbung: Wann und wie?

Die neue erweiterte Klassen steht zur Originalklasse in einem

ist-ein / ist-eine-Art-von

Verhältnis (siehe Seite 7).

Beispiel:

Ein Punkt im 3D-Raum **ist-ein** Punkt.

Ein 3D-Punkt hat alle Eigenschaften eines Punktes.

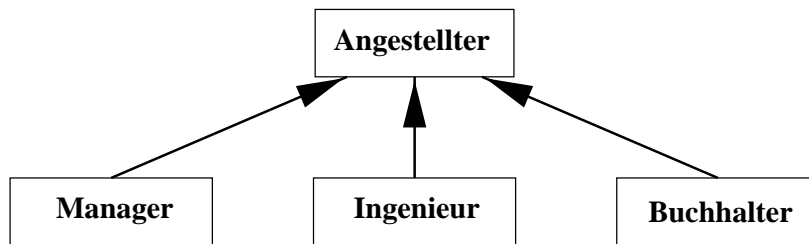
Ein Kreis **ist-kein** Punkt.

Ein Kreis kann aber durch einen Mittelpunkt und einen Radius beschrieben werden.

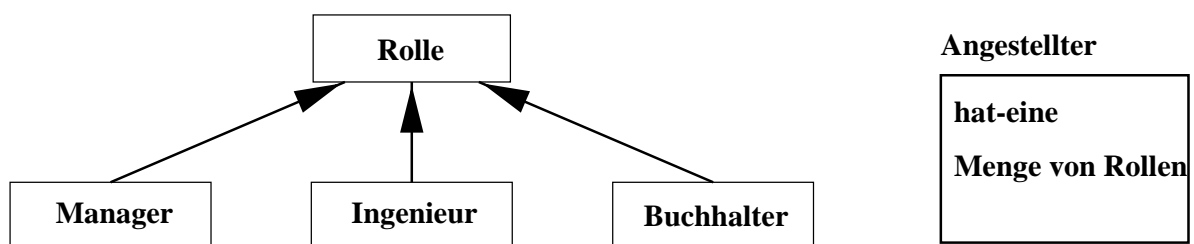
Also: Ein Kreis **hat-ein** Punkt (Mittelpunkt) und einen Radius (*Wiederverwendung durch Komposition, Seite 7*).

So leicht ist es in der Praxis nicht immer!

Beliebtes Beispiel, das in der Praxis scheitert:



Meist kann eine Person mehrere Rollen spielen. Die richtige Lösung ist daher:



4.2 Oberklassen-Konstruktion

```

public point3D(float x, float y, float z)
{
    super(x, y);
    this.z = z;
}
  
```

Regeln:

- `super` ist die erste Anweisung im Konstruktor.
- Wird `super` nicht explizit angegeben, wird der Default-Konstruktor `super ()` aufgerufen. Compiler-Fehler, wenn die Oberklasse keinen Default-Konstruktor hat.
- Konstruktoren, die durch `this` andere Konstruktoren aufrufen, enthalten keinen `super`-Aufruf (siehe Seite 47).

Auf diese Art erhalten wir eine

Konstruktor-Kette

die bei jeder Objekt-Erzeugung bis zur Wurzelklasse `Object` hinaufreicht.

Im Gegensatz dazu erhalten wir **nicht** automatisch eine

Destruktor-Kette

Diese müssen wir bei Bedarf explizit aufbauen:

```
finalize()  
{  
    ...  
    super.finalize();  
}
```

Dies ist durchaus ratsam (Mehr zu `finalize` auf Seite 49)

4.3 Was wird vererbt?

Eine Klasse umfaßt folgende Elemente

- die von der direkten Superklasse geerbten Elemente (außer für `Object`).
- die von allen direkten Super-Interfaces geerbten Elemente.
- Elemente, die im Rumpf der Klasse deklariert sind.

Hierbei gelten als Elemente

- Attribute (“Felder”)
- Methoden

Keine Elemente im Sinne dieser Definition sind also

- Konstruktoren
- Initialisierungsblöcke
- innere Klassen

Geerbt werden alle Elemente der direkten Oberklassen und Ober-Interfaces, die in einer Klasse

- zugreifbar sind
- nicht verdeckt sind
- nicht überschrieben sind

Wegen der “Zugreifbarkeits-Klausel” gilt insbesondere

- Eine Klasse **erbt keine** privaten Elemente der Oberklasse.
- Eine Klasse **erbt keine** Elemente der Oberklasse, die Paketsichtbarkeit haben, aber zu einem anderen Paket gehören.

4.4 Verdecken von Attributen

Ein Attribut-Deklaration (statisch oder nicht-statisch) **verdeckt** eine Attribut-Deklaration einer Oberklasse oder eines Ober-Interfaces, die den gleichen Namen hat.

Der Typ von verdeckendem und verdeckten Attribut muß dabei **nicht** der gleiche sein.

```
class Point
{ public float x,y;
  static int pcount = 0;
}
class Test extends Point
{ static int x;
  boolean pcount;
}
```

Es gibt drei Möglichkeiten, aus der Unterklasse auf verdeckte Namen zuzugreifen:

- bei Klassenvariablen: durch den qualifizierten Namen:
`Point.pcount`
- durch `super`:
`super.x` und `super.pcount`
- durch Typ-Casts:
`((Point)this).x` und `((Point)this).pcount`

4.5 Verdecken durch Klassenmethoden

Eine Klassenmethode (siehe Seite 39) **verdeckt** alle Methoden gleicher Signatur in Oberklassen und Ober-Interfaces, die ansonsten zugreifbar wären (d.h. z.B. nicht die privaten Methoden).

Eine Klassenmethode darf allerdings keine Instanzmethode verdecken!

Wie bei den Attributen gibt es drei Möglichkeiten, aus der Unterklasse auf verdeckte Klassenmethoden zuzugreifen:

- durch den qualifizierten Namen
- durch `super`:
- durch Typ-Casts zum Typ der Oberklasse

4.6 Überschreiben durch Instanzmethoden

Eine Instanzmethode überschreibt alle Methoden gleicher Signatur in Oberklassen und Ober-Interfaces, die ansonsten zugreifbar wären (d.h. z.B. nicht die privaten Methoden).

Eine Instanzmethode darf allerdings keine Klassenmethode überschreiben!

Die einzige Möglichkeit, aus der Unterklasse auf überschriebene Instanzmethoden zuzugreifen ist der

super-Methodenaufruf

```
class Ober
{ String WhoAreYou()
  { return "Ober";
  }
}
class Unter extends Ober
{ String WhoAreYou()
  { return "Unter";
  }

  void identify()
  { System.out.println(
    "I'm " + WhoAreYou() +
    ". My superclass is " +
    super.WhoAreYou() + ".");
    Ober o = (Ober)this;
    System.out.println(
      "My superclass is not " +
      o.WhoAreYou() + ".");
  }
}
```

Die Anweisung `new Unter().identify();` schreibt:

```
I'm Unter. My superclass is Ober.
My superclass is not Unter.
```

4.7 Zusätzliche Regeln für Verdeckung und Überschreiben von Methoden

- Wenn eine Methoden-Deklaration eine andere verdeckt oder überschreibt, dann ist es ein Fehler, wenn sie einen unterschiedlichen Ergebnistyp haben.
- Eventuelle `throws`-Klauseln in verdeckenden oder überschreibenden Methoden dürfen nicht im Konflikt zu denen der verdeckten oder überschriebenen Methoden stehen (dazu später mehr).

- Die Zugriffsspezifikation in verdeckenden oder überschreibenden Methoden müssen mindestens so offen sein wie die der verdeckten oder überschriebenen Methoden.

4.8 Dynamische Bindung

In polymorphen Sprachen kann bei überschriebenen Methoden erst zur Laufzeit festgestellt werden, welche Methode aufgerufen wird (*dynamic method lookup*):

```
class UnterUndOber
{
    Ober a[] = new Ober[100];
    for (int i = 0; i < a.length; i++)
        a[i] = prim(i) ?
            new Ober() :
            new Unter();
    for (int i = 0; i < a.length; i++)
        a[i].WhoAreYou();
}
```

Der Unterschied zwischen Überschreiben / Verdecken

Obwohl in mancher Hinsicht ähnlich, gibt es zwischen überschriebenen Methoden und verdeckten Attributen oder Klassenmethoden wichtige Unterschiede:

- verdeckte Felder erreicht man durch einfachen Typ-Cast.
- überschriebene Methoden werden grundsätzlich dynamisch identifiziert. Typ-Cast bewirkt nichts.
- nur `super` unterbindet die dynamische Bindung.

```
class Ober
{
    String id = "ooo";
    String WhoAreYou()
}
```

```
        { return "Ober";  
        }  
    }  
class Unter extends Ober  
{ String id = "uuu";  
  String WhoAreYou()  
  { return "Unter";  
  }  
  
void identify()  
{ Unter u = new Unter();  
  Ober o = u;  
  System.out.println(o.id);  
  System.out.println(u.id);  
  System.out.println(o.WhoAreYou());  
  System.out.println(u.WhoAreYou());  
}
```

Ausgabe von `new Unter().identify()`:

```
ooo  
uuu  
Unter  
Unter
```

4.9 Überladen von Methoden

Zwei Methoden einer Klassen heißen **überladen**, wenn sie den gleichen Namen aber unterschiedliche Signatur haben.

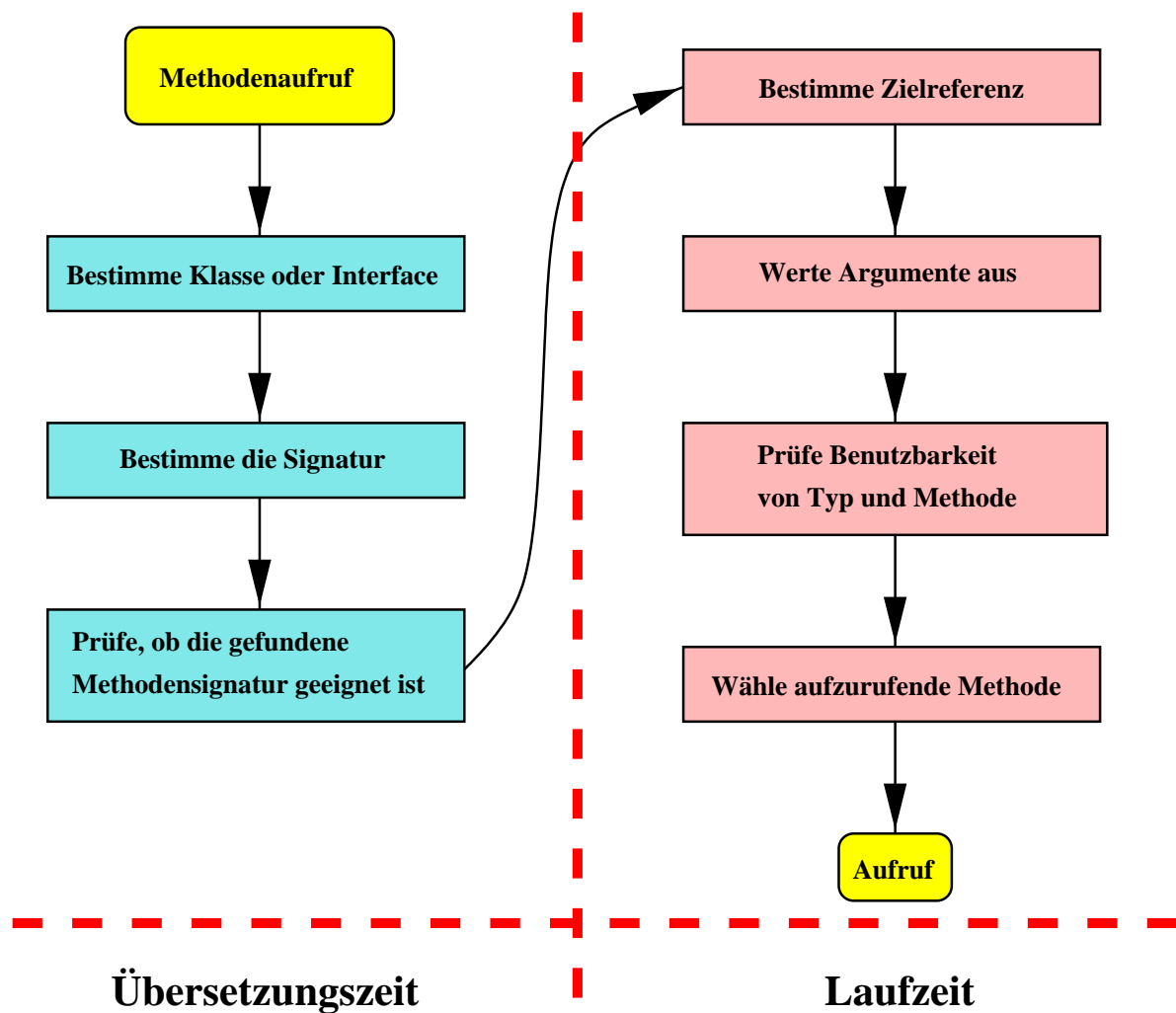
Es spielt keine Rolle,

- ob die Methoden in der Klasse deklariert oder geerbt sind.
- wie der Ergebnistyp der Methoden aussieht.

Erbt man überladene Methoden, kann man einzelne davon überschreiben, ohne die nicht überschriebenen zu verlieren (Unterschied zu C++).

Für einen Methodenaufruf wird die Signatur der aufzurufenden Methode zur Übersetzungszeit festgelegt; die aufzurufende Methode zur Laufzeit.

Grober Ablauf der statischen und dynamischen Methodenbindung



Der Ablauf der statischen und dynamischen Methodenbindung im einzelnen

Bestimme Klasse oder Interface

Wie heißt die aufzurufende Methode und in welcher Klasse (oder welchem Interface) suchen wir die passende Methoden-Definition?

- Methodenaufruf `Methodenname (. . .)`, wobei `Methodenname` drei Formen haben kann:
 - Einfacher Bezeichner: Die Methode heißt `Bezeichner` und die Suchklasse ist die, die den Aufruf enthält.
 - Qualifizierter Name `TypName.Bezeichner`: Die Methode heißt `Bezeichner` und die Suchklasse ist `TypName`. Es ist eine Klassenmethode, daher darf `TypName` kein Interface sein.
 - Sonst ist die Form `FeldName.Bezeichner`: Die Methode heißt `Bezeichner` und die Suchklasse ist der deklarierte Typ von `FeldName`.
- Methodenaufruf `Ausdruck.Bezeichner (. . .)`: Die Methode heißt `Bezeichner` und die Suchklasse ist der Typ von `Ausdruck`.
- Methodenaufruf `super.Bezeichner (. . .)`: Die Methode heißt `Bezeichner` und die Suchklasse ist die Oberklasse der Klasse, die den Aufruf enthält.

Bestimme die Signatur

Welche Methodendeklarationen der Suchklasse sind anwendbar und zugreifbar? Welche von denen ist die speziellste?

Eine Methodendeklaration ist **anwendbar** auf einen Methodenaufruf, wenn

- die Anzahl der Parameter übereinstimmt
- die Typen der aktuellen Parameter verträglich sind mit denen der formalen, d.h.:
 - die Typen sind gleich
 - die Typen der aktuellen Parameter sind durch **Grundtyp-Ausweitung** anpaßbar.
 - die Typen der aktuellen Parameter sind durch **Referenztyp-Ausweitung** anpaßbar.

Eine Methodendeklaration ist **zugreifbar** für einen Methodenaufruf, wenn die Zugriffsrechte (Seite 64) es erlauben.

Gibt es mehrere anwend- und zugreifbare Methodendeklarationen, nimm die speziellste (die mit den “spezialisiertesten” Parametertypen):

```
void test (Ober o1, Ober o2) { ... }  
void test (Unter u1, Ober o1) { ... }  
void test (Ober o1, Unter u1) { ... }  
void test (Unter u1, Unter u2) { ... }
```

Compiler-Fehlermeldungen gibt es, wenn

- es keine anwend- und zugreifbare Methodendeklaration gibt
- wenn nicht eindeutig ist, welche Methodendeklaration die speziellste ist.

**Prüfe, ob die gefundene
Methodensignatur geeignet ist**

Wir haben jetzt die **Übersetzungszeit-Deklaration** der aufzurufenden Methode. Die muß noch 3 Prüfungen bestehen:

- Wenn der Aufruf die Form `Bezeichner` hat und in statischem Kontext (Klassenmethode , statischer Initialisierungsblock oder Initialisierer einer Klassenvariablen) erfolgt, muß die Übersetzungszeit-Deklaration statisch sein.
- Wenn der Aufruf die Form `TypName . Bezeichner` hat, muß die Übersetzungszeit-Deklaration statisch sein.
- Wenn die Übersetzungszeit-Deklaration den Typ `void` hat, darf sie nur in Kontexten benutzt werden, wo kein Wert erwartet wird.

Bestimme Zielreferenz

Welches Objekt soll die Methode ausführen?

- Methodenaufruf `Methodenname(...)`, wobei `Methodenname` drei Formen haben kann:
 - Einfacher Bezeichner: Wenn Klassenmethode, dann keine Zielreferenz, sonst der Wert von `this`.
 - Qualifizierter Name `TypName.Bezeichner`: Es ist eine Klassenmethode, daher keine Zielreferenz.
 - Form `FeldName.Bezeichner`: Wenn Klassenmethode, dann keine Zielreferenz, sonst der Wert von `FeldName`.
- Methodenaufruf `Ausdruck.Bezeichner(...)`: Wenn Klassenmethode, dann keine Zielreferenz: werte `Ausdruck` aus, aber verwirfe Ergebnis. Sonst werte `Ausdruck` aus und benutze Ergebnis als Zielreferenz.
- Methodenaufruf `super.Bezeichner(...)`: Die Zielreferenz ist der Wert von `this`

Werte Argumente aus

Die Argumentausdrücke werden nacheinander von links nach rechts ausgewertet.

Wenn eine dieser Auswertungen abbricht, wird keines der Argumente rechts davon ausgewertet (bzw. man merkt nichts davon) und der gesamte Methodenaufruf wird abgebrochen.

Prüfe Benutzbarkeit von Typ und Methode

Gibt es es die aufzurufende Methode aus der Übersetzungszeitdeklaration überhaupt noch und darf man auf sie zugreifen?

- Wenn die Methode nicht mehr da ist, gibt es einen `NoSuchMethodError`.
- Wenn der Aufruf über ein Interface erfolgte und der Zielreferenztyp dieses Interface nicht mehr implementiert, gibt es einen `IncompatibleClassChangeError`.

- Wenn sich die Zugriffsrechte so verändert haben, daß sie keinen Zugriff mehr erlauben, gibt es einen `IllegalAccessError`.

Der Interpretierer merkt sich das Ergebnis der Prüfung solange die betreffende Klasse geladen bleibt.

Wähle aufzurufende Methode

Welche Methode soll ausgeführt werden?

- Der Aufruf war **statisch**: Die aufzurufende Methode ist die, die der Übersetzer bestimmt hat.
- Wenn eine Zielreferenz da ist und diese den Wert `null` hat: `NullPointerException`.
- Wenn eine Zielreferenz da ist:
 - der Aufruf ist **nicht virtuell**, d.h. kein Überschreiben, keine dynamische Bindung : Die aufzurufende Methode ist die, die der Übersetzer bestimmt hat
 - Der Aufruf ist **virtuell, interface**: Die aufzurufenden Methode bestimmt sich aus dem Laufzeit-Typ der Zielreferenz.
 - Der Aufruf geschieht über **super**: Die aufzurufenden Methode bestimmt sich aus dem Ober-typ der Klasse, die den Aufruf enthält.

4.10 Die `final`-Kennzeichnung

Als `final` können in Java-Programmen markiert werden:

- Attribute
- lokale Variablen
- Parameter (Java1.1)
- Methoden
- Klassen

Bedeutung in allen Fällen: Dieses Element kann nicht verändert werden.

4.10.1 Finale Daten

Final kennzeichnet konstante Daten, also sowohl

Übersetzungszeitkonstanten

```
final int MAX = 12;
```

also auch

Laufzeitkonstanten

```
final int RAND = (int)random*12;
```

final-Werte müssen bei ihrer Deklaration initialisiert werden:

```
// konstanter float-Wert:  
static final float PI = 3.14;  
// konstante Referenz:  
final Uhr u = new Uhr(14,50);
```

Ausnahme: Die in Java1.1 neuen **Blanko-Konstanten** (*blank finals*):

Konstanten, die für jede Instanz einer Klasse einen anderen Wert haben können. Jeder Konstruktor muß diesen Wert zuweisen:

```
class Blankfinaltest  
{ final int C;  
  int val;  
  Blankfinaltest()  
  { C = 7;  
  }  
  Blankfinaltest(int v) // FALSCH  
  { val = v;  
  }  
}
```

Der Compiler:

Blank final variable 'C' may not have been initialized. It must be assigned a value in an initializer, or in every constructor.

4.10.2 Finale Methoden

Finale Methoden können von abgeleiteten Klassen nicht überschrieben werden:

```
class Finalm
{ final void m()
  { System.out.println("fini");
  }
}

class Finalsub extends Finalm
{ void m() // FALSCH
  { System.out.println("over");
  }
}
```

Der Compiler:

Final methods can't be overridden. Method void m() is final in class Finalm.

Gründe für finale Methoden:

- Sicherheit: Niemand kann das Verhalten dieser Methode ändern
- Effizienz: Die Methodenbindung kann für finale Methoden zur Übersetzungszeit festgelegt werden (keine dynamische Bindung).

Insgesamt kann der Java-Compiler in folgenden Fällen dynamische Methodenbindung durch einen schnellen Direktaufruf ersetzen:

- final-Methoden
- Methoden aus final-Klassen

- private-Methoden
- Klassenmethoden

Darüberhinaus kann in diesen Fällen **Inlining** der Methodenrumpfe in Betracht gezogen werden.

4.10.3 Finale Klassen

Durch `final`-Kennzeichnung einer gesamten Klasse verbietet man die Erweiterung dieser Klasse:

```
final class Finalc
{ void m()
  { System.out.println("fini");
  }
}

class Finalsub extends Finalc // FALSCH
{ void m()
  { System.out.println("over");
  }
}
```

Der Compiler:

Can't subclass final classes: class Finalc

Die Gründe für finale Klassen sind ebenfalls Sicherheit und/oder Effizienz.

4.11 Abstrakte Methoden und abstrakte Klassen

Eine Methode, von der nur die Signatur und nicht der Rumpf definiert ist³. Erbende Klassen können die Implementierung “nachliefern”.

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

³In C++ heißen abstrakte Methoden “pur virtuell”

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

class Benchmark extends Bench
{ void benchmark()
  { // leere Methode
  }
  public static void main (String[] args)
  { int count = Integer.parseInt(args[0]);
    long time =
      new Benchmark().repeat(count);
    System.out.println(count +
                        " methods in " +
                        time +
                        " milliseconds");
  }
}
```

Regeln für abstrakte Methoden und Klassen:

- Jede Klasse, die eine abstrakte Methode hat, ist eine **abstrakte Klasse** und muß als solche markiert werden.
- Von abstrakten Klassen lassen sich keine Instanzen bilden.
- Eine Klasse kann auch dann als abstrakt markiert werden, wenn sie keine abstrakte Methoden hat. So verhindert man Instantiierung.
- Von Unterklassen von abstrakten Klassen können Instanzen gebildet werden, wenn sie
 - alle geerbten abstrakten Methoden überschreiben
 - und dafür Implementierungen liefern.

Geschieht dies nicht, ist die Unterklasse selbst abstrakt.

Typisches Anwendungsmuster: Eine Klasse bezieht “Expertenwissen” oder spezielles Verhalten von ihren Unterklassen.

Das Design der Klasse wird angegeben, nicht aber die (vollständige) Implementierung.

5 Interfaces

Klassen beschreiben Entwurf **und** Implementierung.

**Interfaces beschreiben den
reinen Entwurf.**

Interfaces enthalten

- statische Konstanten
- abstrakte Methoden

Beispiel:

```
interface Verbose
{ int SILENT = 0;
  int VERBOSE = 1;
  void setVerbosity(int level);
}
```

Anwendung:

```
class VerboseThing extends Thing
  implements Verbose
{ ...
}
```

5.1 Beispiel einer Interface-Anwendung

Aufgabe: Objekten von benutzerdefinierten Klassen sollen ein oder mehrere benannte Attribute zugeordnet werden.

z.B.

```
("Farbe", #ff00ff)
("transparent", false)
```

Zunächst: Der Typ Attr

```
public class Attr
{ private String name;
  private Object value;
  public Attr(String name)
  { this.name = name;
  }
  public Attr(String name, Object value)
  { this.name = name;
    this.value = value;
  }
  public String nameOf()
  { return name;
  }
  public Object valueOf()
  { return value;
  }
  public Object valueOf(Object newValue)
  { Object oldValue = value;
    value = newValue;
    return oldValue;
  }
}
```

Idee:

Ein Typ Attributed mit folgender Funktionalität:

```
void add(Attr newAttr)
```

```
Attr find(String AttrName)
Attr remove(String AttrName)
```

Benutzertypen könnten diesen Typ `Attributed` erweitern und wären somit attribuiert:

```
class Figure extends Attributed
{ ...
}
```

Problem: Einfacherbung in Java verbietet dies im Normalfall, da der Benutzertyp bereits eine Oberklasse hat.

Einzigster Ausweg: Einbau der gewünschten Funktionalität in die Wurzelklasse `Object`. Geht nicht und ist auch nicht sinnvoll!

Lösung mit Interfaces:

```
public interface Attributed
{ void add(Attr newAttr);
  Attr find(String attrName);
  Attr remove(String attrName);
}

class AttributedFigure extends Whatever
  implements Attributed
{ ...
}
```

Wir haben eine klare und einfache Lösung unseres Problems, können allerdings die Implementierung von `Attributed` nicht erben, sondern müssen sie selbst liefern.

zum Beispiel so:

```
import java.util.*;

public class AttributedImpl
```



```
    implements Attributed
{
    protected Hashtable attrTable
        = new Hashtable();

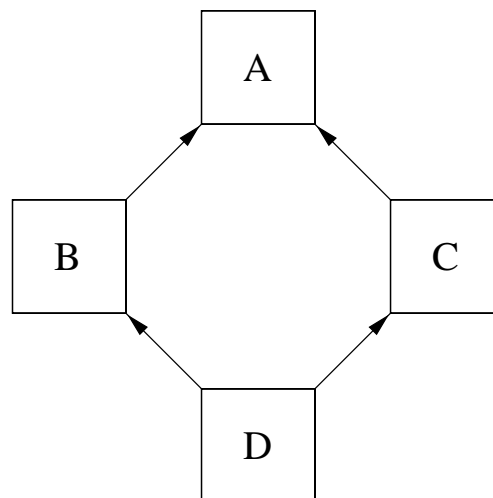
    public void add(Attr newAttr)
    {
        attrTable.put(newAttr.nameOf(), newAttr);
    }

    public Attr find(String name)
    {
        return (Attr) attrTable.get(name);
    }

    public Attr remove(String name)
    {
        return (Attr) attrTable.remove(name);
    }
}
```

5.2 Einfacherbung gegen Mehrfacherbung

Mehrfacherbung von Klassen führt bei rautenförmiger Hierarchiestruktur (*diamond inheritance*)



zu Mehrdeutigkeiten.

Ursache für diese Mehrdeutigkeiten ist der Zugriff auf Implementierungsaspekte (Attribute)

Also: Einfacherbung für Klassen

Mehrfacherbung von Entwurfsaspekten macht keine Probleme

Also: Mehrfacherbung für Interfaces

```
class AttributedFigure extends Whatever
    implements Attributed, Verbose
{ ...
}
```

5.3 Interface-Deklarationen

Ein Interface kann `public`- oder Paket-Sichtbarkeit haben (default). Bedeutung genau wie bei Klassen.

```
public interface Verbose
{ ...
}
```

Ein Interface ist abstrakt. Dies kann man durch `abstract` kennzeichnen, ist aber überflüssig.

Ein Interface kann ein oder mehrere Ober-Interfaces (*Superinterfaces*) haben:

```
interface X
    extends Y, Z
```

```
{ ...  
}
```

Die Superinterface-Beziehung darf natürlich weder direkt noch indirekt zyklisch sein.

Es gibt für Interfaces keine Analogie zur Wurzelklasse `Object`.

Interface-Elemente sind

- statische Konstanten
- abstrakte Methoden

Diese sind implizit `public`.

Weder `public` noch `static` und `final` für die Konstanten brauchen spezifiziert zu werden.

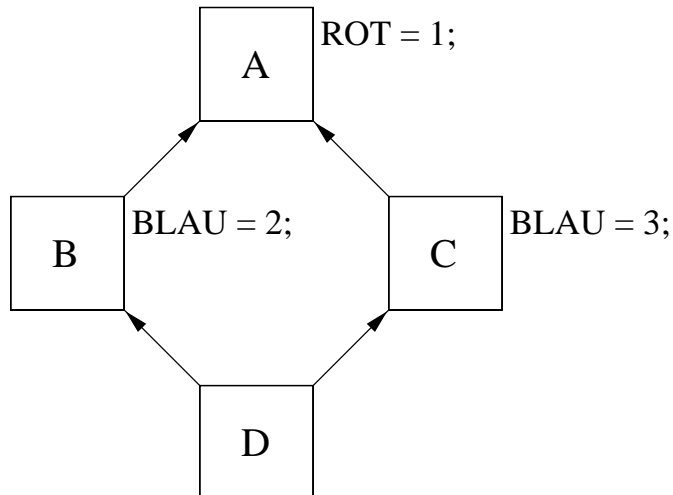
5.4 Konstanten in Interfaces

Die Konstanten müssen initialisiert werden.

```
interface Verbose  
{ int SILENT = 0;  
  int VERBOSE = SILENT+1;  
  void setVerbosity(int level);  
}
```

Initialisierungsausdrücke werden ausgewertet, wenn das Interface geladen wird. Es gelten die gleichen Regeln wie bei der Initialisierung von Klassenvariablen (Reihenfolge, kein `this`, etc., siehe Seite 42).

5.5 Mehrdeutigkeiten bei Konstanten



D erbt ROT mehrmals: Kein Problem.

D erbt verschiedene Felder namens BLAU: Kein Problem, wenn D BLAU nicht benutzt, ansonsten Fehlermeldung.

5.6 Methoden in Interfaces

Implizit `public` und `abstract`. Verboten sind die Spezifikatoren `static` und `final`, wobei die Methode die eine Interface-Methode implementiert durchaus `final` sein darf.

Interface-Methoden können Methoden gleicher Signatur aus Ober-Interfaces **überschreiben**. Fehler, wenn Ergebnistyp verschieden.

Ein Interface kann 2 Methoden gleicher Signatur erben. Kein Problem, wenn der Ergebnistyp gleich ist.

Zwei Interface-Methoden mit gleichem Namen aber verschiedener Signatur sind **überladen**. Ergebnistyp spielt dabei keine Rolle.

Aus “Streifzug durch Java” (Seite 29):

Interfaces erweitern die Möglichkeiten der Polymorphie:

Die **Obertypen** einer Klasse T umfassen

1. den Typ, den T erweitert (`extends`)
2. die Typen, die T implementiert (`implements`)
3. die Obertypen der Typen aus [1.] und [2.].

Ein Objekt vom Typ T darf überall dort verwendet werden, wo Obertypen von T zulässig sind.

5.7 Beispiele für Interfaces der Java-Bibliothek

public interface Cloneable⁴

Klassen, die dieses Interface implementieren signalisieren der Methode `Object.clone()`, daß ihre Objekte kopiert werden können (elementweise Kopie).

`clone()`-Aufrufe für Objekte, deren Klassen dieses Interface nicht implementieren, führen zu einer `CloneNotSupportedException`.

public interface Runnable

Jede Klasse, deren Instanzen als Thread ausgeführt werden sollen, sollte Runnable implementieren.

Method Index

```
public abstract void run()
```

⁴Ist übrigens falsch geschrieben. Richtig wäre “Clonable”.

When an object implementing interface `Runnable` is used to create a thread, starting the thread causes the object's `run` method to be called in that separately executing thread.

6 Arrays

Arrays in Java haben vieles mit Objekten gemeinsam:

- Arrays werden über Referenzen angesprochen.
- Arrays werden dynamisch erzeugt.
- Arrays unterliegen der Garbage Collection .
- Arrays können an Variablen vom Typ `Object` zugewiesen werden.

6.1 Erzeugung von Arrays

Mit new:

```
Button buttons[] = new Button[12];
```

oder

```
Button[] buttons = new Button[12];
```

Dies ist **kein** Konstruktoraufruf. Es werden keine `Button`-Objekte erzeugt.

Das Array wird mit den Default-Initialwerten (Seite 46) initialisiert (hier `null`).

Mit Initialisierer:

```
int[] lookup = {1, 2, 4, 8, 16};
```

erzeugt dynamisch ein Array geeigneter Länge und initialisiert dessen Elemente. Ausdrücke beliebig (anders als in C).

Seit Java1.1:

Anonyme Arrays durch Kombination Initialisierer/new:

```
mymenu(new String[] { "Open",  
                      "Close",  
                      "Exit"  
});
```

6.2 Mehrdimensionale Arrays

Realisiert als Arrays von Arrays (wie in C).

```
byte[][] twodimarray =  
    new byte[256][16];
```

oder

```
byte twodimarray[][] =  
    new byte[256][16];
```

oder gar

```
byte[] twodimarray[] =  
    new byte[256][16];
```

Ablauf:

- 1.) Allokiere ein 256-elementiges Array (Typ `byte[][]`).
- 2.) Allokiere 256 16-elementige Arrays (Typ `byte[]`).
Setze Referenzen auf diese in das Array aus 1.).
Initialisiere die 16 Bytes jeweils mit 0.

Teilweise Allokation:


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

“Hintere” Dimensionen dürfen offengelassen werden.

Nützlich für “nicht-rechteckige” Arrays:

```
short[][] dreieck = new short[10][];  
for (int i=0; i < dreieck.length; i++)  
    dreieck[i] = new short[i+1];
```

Allokation mit geschachtelten Initialisierern:

```
int[][] a3x2 = {{1,2}, {2,3}, {3,4}};
```

Geht auch “nicht-rechteckig”:

```
int[][] b = {{1,2}, {2,3,4}, {5, 6}};
```

6.3 Zugriff auf Arrays

Arrays haben ein konstantes Attribut `length`, das die Anzahl der Elemente enthält.

```
for (int i=0; i < dreieck.length; i++)  
    dreieck[i] = new short[i+1];
```

Die Array-Größe ist nicht Eigenschaft des Typs. Eine `String[]`-Variable, zum Beispiel, kann unterschiedliche Array-Objekte aufnehmen:

```
String[] strings;  
strings = new String[10];  
strings = new String[42];
```

7 Grundsymbbole, Operatoren und Ausdrücke

In diesem Kapitel geht es um den grundlegenden Aufbau von Java-Programmen.

7.1 Unicode-Buchstaben

Java benutzt nicht 8-bit Zeichensätze wie ASCII oder EBCDIC sondern den 16-Bit Zeichensatz **Unicode**.

Die ersten 256 Zeichen des Unicode sind identisch mit der ASCII-Variante *Latin-1*.

Die meisten anderen Zeichen sind mit unseren Editoren nicht darstellbar, daher gibt es eine Escape-Schreibweise:

```
\uddd      (d ist Hexadezimalziffer)
```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
4																
5																
6																
7																
8																
9																

U+4E00 - U+4EFF

To change to another block,
click on one of the boxes to the
left.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	一	丂	北	丰	乂	乐	习	买	龜	亏	亠	京	什	仝	仵	仰
1	丁	丑	兩	𠂇	𠂇	𠂇	乡	乱	乾	云	亡	宿	仁	仝	仵	伶
2	𠂇	刃	丟	串	乂	兵	𠂇	盜	亂	互	亢	亲	𠂇	仝	仵	仲
3	七	专	𠂇	弗	乃	兵	𠂇	乳	𠂇	𠂇	𠂇	毫	仵	仝	代	𠂇
4	上	且	兩	临	乂	乔	𠂇	𠂇	𠂇	五	交	𠂇	𠂇	仵	令	𠂇
5	丁	丕	严	𠂇	久	𠂇	𠂇	𠂇	丁	井	亥	𠂇	𠂇	仵	以	𠂇
6	𠂇	世	並	、	久	乖	书	𠂇	了	三	亦	𠂇	𠂇	仵	𠂇	𠂇
7	万	𠂇	喪	𠂇	𠂇	乘	𠂇	𠂇	𠂇	𠂇	𠂇	廉	𠂇	𠂇	𠂇	𠂇
8	丈	丘	𠂇	丸	么	乘	𠂇	𠂇	𠂇	𠂇	亨	𠂇	𠂇	𠂇	𠂇	𠂇
9	三	丙	𠂇	丹	乂	乙	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇
A	上	业	个	为	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇
B	下	丛	𠂇	主	之	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇
C	𠂇	东	𠂇	井	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇
D	不	丝	中	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇
E	与	丞	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇
F	𠂇	丟	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇	𠂇

7.2 Kommentare

Drei Kommentar-Stile

- Zeilenkommentare: vom // bis zum Zeilenende
- Klammerkommentare: zwischen /* und */.
- Dokumentations-Kommentare: zwischen /** und */.

Siehe Seite 21.

7.3 Bezeichner

Buchstabe (einschließlich _ und \$), gefolgt von Buchstaben und Ziffern.

Wegen Unicode umfassen Buchstaben und Ziffern jedoch alle erdenklichen Zeichen fast aller geschriebenen Sprachen.

7.4 Reservierte Wörter

abstract	double	int	static
boolean	else	interface	super
break	extends	long	switch
byte	final	native	synchronized
case	finally	new	this
catch	float	null	throw
char	for	package	throws
class	goto *	private	transient *
const *	if	protected	try
continue	implements	public	void
default	import	return	volatile
do	instanceof	short	while

Mit * markierte Schlüsselwörter sind reserviert aber (noch) nicht verwendet.

7.5 Grundtypen

Ganze Zahlen:

byte	8-bit 2er Komplement	Byte
short	16-bit 2er Komplement	Kurz-Integer
int	32-bit 2er Komplement	Integer
long	64-bit 2er Komplement	Lang-Integer

Fließpunktzahlen

float	32-bit IEEE 754	Einfachpräzision
double	64-bit IEEE 754	Doppelpräzision

Andere Typen

char	16-bit Unicode-Zeichen	Buchstabe
boolean	wahr oder falsch	Boolescher Wert

7.6 Literale

Ganze Zahlen:

Oktaldarstellung	052
Dezimaldarstellung	42
Hexadezimaldarstellung	0x2A

Fließpunktzahlen

Vier Schreibweisen:

```

Digits . [Digits] [Expo] [FloatSuffix]
      z.B. 3.1E4
. Digits [Expo] [FloatSuffix]
      z.B. .1F
Digits Expo [FloatSuffix]
      z.B 17e12
Digits [Expo] FloatSuffix
      z.B. 0F

```

Ohne “FloatSuffix” ist der Typ `double`.

Buchstaben

Buchstaben-Literale werden in einfache Anführungszeichen eingeschlossen: `'p'`.

Die folgenden Escape-Sequenzen sind definiert:

<code>\n</code>	newline	<code>\t</code>	Tabulator
<code>\b</code>	backspace	<code>\r</code>	return
<code>\f</code>	form feed	<code>\\</code>	der Backslash selbst
<code>\'</code>	single quote	<code>\"</code>	double quote
<code>\ddd</code>	Oktalwertdarstellung		

Boolesche Werte

entweder `true` oder `false`.

7.7 Deklaration von Variablen

Eine Deklaration legt Zugriffsbeschränkungen, Typ und andere Eigenschaften eines Bezeichners fest.

Form:

Modifier Type Identifier_List

Modifier sind

```
public
private      static      synchronized
protected
```

Diese Reihenfolge wird empfohlen, im Prinzip ist sie aber beliebig.

Es gibt 7 Arten von Variablen:

- Klassen-Variablen
- Instanz-Variablen
- Array-Elemente (unbenannt)
- Methoden-Parameter
- Konstruktor-Parameter
- Exception-Handler-Parameter
- Lokale Variablen

7.8 Initialisierung

Wenn der Initialisierungsausdruck bei der Deklaration fehlt, bekommen Attribute Default-Werte:

Typ	Default
boolean	false
char	\u0000
byte	0
short	0
int	0
long	0
float	0.0
double	0.0
Klasse, Interface, Array	null

Lokale Variablen in Methoden, Konstruktoren und Initialisierungsblöcken bekommen keine Default-Werte!

Verwendung führt zu Fehler:

Beispiel:

```
class UnInit
{ static public void main(String av[])
  { double d;
    if (true)
      d=9.9;
    else ;
    System.out.println(d); // FEHLER
  }
}
```

liefert die Compiler-Fehlermeldung

```
Variable d may not have been initialized
```


7.9 Namensanalyse

Durch seine Deklaration lebt jeder Bezeichner in einem bestimmten Namensraum. Alle Bezeichner in einem Namensraum müssen verschieden sein.

Wird ein Bezeichner angewendet, muß festgestellt werden, in welchem Namensraum er lebt um dann auf seine deklarierten Eigenschaften zugreifen zu können.

Der gleiche Bezeichner darf in verschiedenen Namensräumen leben:

```
class Reuse
{ Reuse Reuse(Reuse Reuse)
  { Reuse:
    for (;;)
    { if (Reuse.Reuse(Reuse) == Reuse)
      break Reuse;
    }
    return Reuse;
  }
}
```

Namensräume in Java

- Paket
- Typ
- Methode
- Attribut
- Variable
- Label

Die Namensräume sind geschachtelt, dh. Deklarationen in inneren Strukturen können die Deklarationen in äußeren Strukturen verdecken.

Beispiel:

der Name eines Konstruktorparameters überdeckt den Namen einer Instanzvariable.

```
class MyClass
{ T myattr;
  MyClass (T myattr)
  { this.myattr = myattr;
  }
}
```

Ausnahme:

Die Schachtelung von Blöcken und for-Schleifen kann nicht dazu benutzt werden, Bezeichner aus äußeren Blöcken, for-Schleifen und Methoden-Parameter zu überdecken.

```
class Nesti
{ static public void main(String a[])
{ double d = 9.9;
  for (int i = 0; i < 10; i++)
  { int d; // FEHLER
    boolean a; // FEHLER
    char i; // FEHLER
    ...
  }
}
}
```

liefert

```
'd' is already defined in this method.
'a' is already defined in this method.
'i' is already defined in this method.
```

7.10 Konventionen für die Verwendung von Bezeichnern**Ziel:**

- Lesbarer Code
- Vermeidung von Namenskonflikten und Mißverständnissen

Diese Bezeichnerkonventionen sind Empfehlungen, man muß sich nicht sklavisch daran halten.

So sind z.B. `sin` und `cos` aus der Bibliothek `java.lang.Math` Bezeichner, die gegen die Java-Konventionen verstoßen, sie sind aber üblich.

Klassen und Interfaces

- Substantive, Substantivische Konstruktionen
- Aussagekräftig
- Gemischte Groß-/Kleinschreibung (vorne groß)

Beispiele:

```
ClassLoader  
BufferedInputStream  
HalloWelt
```

Methoden

- Verben, Verbkonstruktionen
- Gemischte Groß-/Kleinschreibung (vorne klein)

Beispiele:

```
sayHello()  
printDataFile()
```

Zusatzregeln:

- Attributzugriff: `getXyz()` und `setXyz()`

- Längenbestimmung: `length()`
- Boolescher Test: `isXyz()`
- Konvertierung: `toString()`

Konstanten

- Wörter getrennt durch `_`
- Großschreibung

Beispiele:

```
MAX_VALUE  
MIN_TEMP
```

Variablen und Parameter

- kurz aber sinnvoll
- Kleinschreibung
- oft Abkürzungen

Beispiele:

```
buf  
cp  
len  
out
```

7.11 Operatorpräzedenz und -assoziativität

Präzedenz beschreibt die “Bindungskraft” von Operatoren.

So ist

$$3 * 5 - 3$$

12 und nicht 6, und

$$(i \geq \text{min} \ \&\& \ i \leq \text{max})$$

prüft ob i zwischen min und max liegt.

Sind zwei Operatoren gleicher Präzedenz in einem Ausdruck benachbart, so bestimmt die **Assoziativität**, welcher von beiden zuerst ausgewertet wird.

$$a + b + c$$

bedeutet (da die Addition linksassoziativ ist)

$$(a + b) + c$$

Die Operatorpräzedenzen absteigend geordnet:

Operator-Art	Operatoren
Postfix-Operator	[] . (params) expr++ expr--
Unärer Operator	++expr --expr +expr -expr ! ~
Erzeugung oder Typ-Cast	new (type) expr
Multiplikation	* / %
Addition	+ -
Shift	<< >> >>>
Relational	< > <= >= instanceof
Gleichheit	== !=
bitweises AND	&
bitweises XOR	^
bitweises OR	
logisches AND	&&
logisches OR	
Bedingung	? :
Zuweisung	= += -= *= /= %= >>= <<=
	>>>= &= ^= =

Alle Wertzuweisungen sind **rechtsassoziativ**, die anderen binären Operatoren sind **linksassoziativ**.

Mit Klammern kann die Operatorbindung beeinflußt werden:

```
while ((v = stream.next()) != null)
{
    ...
}
```

7.12 Auswertungsreihenfolge

Java garantiert, daß die Operanden von Operatoren von links nach rechts ausgewertet werden.

Im Ausdruck

```
x + y + z
```

wird also niemals y vor x oder z vor x oder y ausgewertet.

Auswertungsreihenfolge ist natürlich nur beobachtbar für Operanden mit Seiteneffekten, z.B.

- Zustandsändernde Methodenaufrufe
- Auslösen von Ausnahmen (z.B. Division durch 0)
- Ein-/Ausgabe

Außer bei den Operatoren $\&\&$, $||$ und $?:$ werden stets alle Operanden ausgewertet, bevor eine Operation ausgeführt wird.

7.13 Typanpassung

Typanpassung tritt auf bei Wertzuweisungen, Ausdrücken und Parameterübergaben.

Wir unterscheiden:

- implizite Typanpassung
- explizite Typanpassung

7.13.1 Implizite Typanpassung

Hier unterscheiden wir die Typanpassung für Grundtypen und die für Referenztypen.

Bei Grundtypen wird implizit zwischen zwei Typen konvertiert, wenn der **Bereich** des Zieltyps größer oder gleich ist.

Zum Beispiel:

Zwischen `char` und `int`

Zwischen `long` und `float`

Zwischen `float` und `double`

Bei Referenztypen wird implizit konvertiert zwischen einem Typ `T` und seinen Obertypen.

7.13.2 Explizite Typanpassung

Explizite Casts erlauben Typanpassungen, bei denen der Zieltyp den kleineren Wertebereich umfaßt:

```
double d = 7.88;  
long l = (long) d;
```

Explizite Casts also z.B.

Zwischen `double` und `float`

Zwischen Fließpunkt und Ganzzahl

Von `long` nach `int`, `short`, `byte` oder `char`.

Bei solchen Typanpassungen kann man Genauigkeit verlieren:

```
short s = -134;  
byte b = (byte) s;
```

Die vorderen Bits gehen verloren: `b` hat den Wert 122.

Für Referenztypen erlauben explizite Casts Typumwandlungen entgegen der Klassenhierarchie, also z.B. vom Obertyp zum Untertyp

Solche Typumwandlungen heißen

- Down-Casts
- Typ-Einengung
- Narrowing Conversions
- Unsafe Casting

Sie erfordern einen Laufzeittest, ob das aktuelle Objekt tatsächlich für den Zieltyp zulässig ist. Wenn nicht: `ClassCastException`.

Beispiel:

```
class Ober{}
class Unter extends Ober{}
public class Classcast
{ public static void main(String argv[])
  { Ober objo = new Ober();
    Unter obju = (Unter) objo;
  }
}
```

liefert Laufzeit-Fehler

```
java.lang.ClassCastException: Ober
```

Mögliche Down-Casts:

- Von Klasse `S` nach Klasse `T`, wobei `S` Obertyp von `T` ist.
- Von Klasse `S` nach Interface `K`, falls `S` nicht `final` ist und `K` nicht implementiert.
- Von Klasse `Object` zu einem Array-Typ.
- Von Klasse `Object` zu einem Interface-Typ.
- Von Interface-Typ `K` nach Klasse `T`, falls `T` nicht `final` ist.

- Von Interface-Typ `K` nach Klasse `T`, falls `T final` ist und `K` implementiert.
- Von Interface-Typ `K` nach Interface-Typ `J`, wobei `K` kein Unter-Interface von `J` ist, falls `K` und `J` keine Methode `m` mit gleicher Signatur aber verschiedenem Ergebnistyp deklarieren.
- Von Array-Typ `SC` nach Array-Typ `TC`, falls `SC` und `TC` Referenztypen sind und down-cast von `SC` nach `TC` zulässig ist.

Diese Down-Casts erfordern Laufzeittests, ob der tatsächliche Wert für den Zieltyp zulässig ist.

String Konvertierung

Es gibt Typanpassungen von jedem Typ (einschließlich `Null`) nach `String`.

Verbotene Typanpassungen:

- Von Referenztypen zu Grundtypen.
- Von Grundtypen zu Referenztypen (außer für `Strings`).
- Vom `Null`-Typ zu Grundtypen.
- Zum `Null`typ.
- Zum Typ `boolean`.
- Vom Typ `boolean`, außer `String`-Konvertierung.
- Von Klasse `T` nach Klasse `S`, die nicht in Unter-/Ober-Klassen-Beziehung stehen.
- Von Klasse `S` nach Interface `K`, wenn `S final` ist und `K` nicht implementiert.
- Von Interface `K` nach Klasse `S`, wenn `S final` ist und `K` nicht implementiert.
- Von einer Klasse `T` zu einem Array-Typ, wenn `T` nicht `Object` ist.
- Zwischen zwei Interfaces, die eine Methode gleicher Signatur aber mit verschiedenem Ergebnistyp deklarieren.
- Von Arraytypen nach Typen ungleich `Object` und `String`.
- Von Arraytypen nach Interfaces ungleich `Cloneable`, das von allen Arrays implementiert wird.

- von Array-Typ `SC[]` nach Array-Typ `TC`, wenn es keine erlaubte Anpassung zwischen `SC` und `TC` außer String-Konvertierung gibt.

7.14 Operatoren

7.14.1 Arithmetik

+	<code>op1 + op2</code>	Addiert <code>op1</code> und <code>op2</code>
-	<code>op1 - op2</code>	Subtrahiert <code>op2</code> von <code>op1</code>
*	<code>op1 * op2</code>	Multipliziert <code>op1</code> mit <code>op2</code>
/	<code>op1 / op2</code>	Dividiert <code>op1</code> durch <code>op2</code>
%	<code>op1 % op2</code>	Berechnet den Rest der Division <code>op1</code> durch <code>op2</code>

Außerdem zwei unäre Operatoren

+	<code>+ op1</code>	Unäres Plus (aus Symethrie-Gründen)
-	<code>- op1</code>	Unäres Minus, Negation

Java rechnet ganzzahlig in Zweierkomplement-Arithmetik. Nichts kann über- oder unterlaufen.

Für die Gleitpunkt-Arithmetik wird eine Untermenge des IEEE 754-1985-Standards benutzt. Es gibt ausgezeichnete Werte für $+\infty$, $-\infty$ und NaN (not a number) und Rechenregeln dazu.

7.14.2 String-Konkatenation

`+` wird zur String-Konkatenation verwendet:

```
String boo = "boo";
String cry = boo + "hoo";
cry += "!";
System.out.println(cry);
```

liefert

boohoo!

Die implizite Konvertierung nach `String` gibt es nur im Zusammenhang mit dem Operator `+`:

```
String s;
s = '@' + s + 3.14;
```

aber z.B. nicht bei der Parameterübergabe an `String`-Parameter.

7.14.3 Inkrement/Dekrement

++	op++	Erhöht op um 1, liefert Wert vor Inkrement
++	++op	Erhöht op um 1, liefert Wert nach Inkrement
--	op--	Erniedrigt op um 1, liefert Wert vor Dekrement
--	--op	Erniedrigt op um 1, liefert Wert nach Dekrement

7.14.4 Vergleich und logische Operatoren

Vergleichsoperatoren

>	op1 > op2	op1 ist größer als op2
>=	op1 >= op2	op1 ist größer oder gleich op2
<	op1 < op2	op1 ist kleiner als op2
<=	op1 <= op2	op1 ist kleiner oder gleich op2
==	op1 == op2	op1 und op2 sind gleich
!=	op1 != op2	op1 und op2 sind nicht gleich

Diese Operatoren liefern Boolesche Werte.

Logische Operatoren

&&	op1 && op2	op1 und op2 sind beide true
	op1 op2	op1 oder op2 ist true
!	! op	op ist false

Diese Operatoren werten ihre rechten Operanden nicht aus, wenn der linke bereits das Ergebnis bestimmt (*Boolesche Kurzauswertung*).

Der bedingte Operator ? :

Erlaubt Ausdrücke, die abhängig von einer Bedingung einen von zwei Werten liefern:

```
val = userSetIt ? usersVal : defaultVal;
```

Der Typ eines der Ausdrücke muß dem Typ des anderen Ausdrucks ohne expliziten Cast zuweisbar sein. Der allgemeinere Typ der beiden bestimmt das Ergebnis. Dies gilt auch für Referenztypen.

7.14.5 Bit-Operationen

>>	op1 >> op2	Rechtssshift von op1 um op2 Stellen
>>>	op1 >>> op2	dto. aber von links 0 nachziehen
<<	op1 << op2	Linkssshift von op1 um op2 Stellen
&	op1 & op2	bitweises UND
	op1 op2	bitweises ODER
^	op1 ^ op2	bitweises Exklusiv-Oder
~	~op2	bitweises Komplement

Bitweise Operationen dürfen nur auf ganzen Zahlen und Booleschen Werten ausgeführt werden.

7.14.6 Zuweisungen

=	op1 = op2	
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 &= op2
=	op1 = op2	op1 = op1 op2
^=	op1 ^= op2.	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

8 Anweisungen

Java bietet drei Formen von Anweisungen

- Ausdrucksanweisungen
 - Wertzuweisungen
 - Inkrement/Dekrement mit ++ und --
 - Methodenaufruf (egal ob die Methode einen Wert liefert)
 - Objekterzeugung mit new
- Deklarationsanweisungen
 - deklarieren lokale Variablen
 - irgendwo im Block, nicht nur am Anfang
- Steuerungsanweisungen
 - Beeinflussen den Ablauf der Ausführung (*control flow*).

Mit geschweiften Klammern { und } werden Anweisungen zu **Blöcken** gruppiert. Diese können überall anstelle von einzelnen Anweisungen verwendet werden.

Dieses Kapitel beschäftigt sich mit den Steuerungsanweisungen.

8.1 Verzweigung

Die Formen

```
if (boolean_expression)
    statement
```

und

```
if (boolean_expression)
    statement_1
else
    statement_2
```

Bei geschachtelten `if`-Anweisungen wird ein `else` dem jeweils letzten `else`-losen `if` zugeordnet.

8.2 Fallunterscheidung

Die `switch`-Anweisung wertet einen ganzzahligen Ausdruck aus und verzweigt abhängig von dessen Wert zu einer markierten Anweisung.

Beispiel:

```
switch (Verbosity)
{ case NORMAL:
    System.out.println(summaryState);
    // Weiter in der Anweisungsfolge
  case VERBOSE:
    System.out.println(basicState);
    break;
    // Raus aus dem Switch
  default:
    throw new UnexpectedVerbException
        (Verbosity);
}
```

Wichtig:

- Ohne `break`, `return` oder `throw` geht es weiter mit der nächsten Anweisung im Block (*Fall-through*).
- Wenn kein `case` zutrifft und kein `default` existiert, endet die Fallunterscheidung normal.

8.3 while- und do-Schleifen

Die Formen

```
while (boolean_expression)
    statement
```

und

```
do statement
while (boolean_expression)
```

Wichtig:

- Die while-Schleife prüft die Bedingung vor Ausführung des Rumpfes.
- Die do-Schleife prüft die Bedingung nachdem der Rumpf ausgeführt wurde.

8.4 for-Schleife

Form:

```
for (init_expr; boolean_expr; incr_expr)
    statement
```

entspricht in der Wirkung der while-Schleife:

```
init_expr;
while (boolean_expr)
{
    statement
    incr_expr;
}
```

mit dem Unterschied, daß `incr_expr` immer ausgeführt wird, wenn während der Ausführung von `statement` eine `continue`-Anweisung (siehe Seite 123) ausgeführt wird.

8.5 Abbruch-Anweisungen: break, continue, return

Die Abbruch-Anweisungen break und continue können Label als Argumente benutzen, die die Aufsetzpunkte angeben.

Zur Spezifikation solcher Aufsetzpunkte gibt es markierte Anweisungen

```
label: statement
```

8.5.1 Die break-Anweisung

Zwei Formen:

```
break
```

und

```
break identifier
```

Das break **ohne** Label verläßt die umgebende while-, do-, for- oder switch-Anweisung. Existiert die nicht, gibt es einen Übersetzerfehler.

Das break **mit** Label verläßt die mit dem Label markierte umgebende Anweisung (z.B. auch einen Block). Existiert die nicht, gibt es einen Übersetzerfehler.

Beispiel:

```
class Breaktry
{ private float [][] Matrix;
  boolean find(float val)
  { boolean found = false;
    int x, y;
    search:
    for (y = 0; y < Matrix.length; y++)
      for (x = 0;
           x < Matrix[y].length;
           x++)
```

```
        if (Matrix[y][x] == val)
        { found = true;
          break search;
        }
    if (!found)
        return false;
    // mach irgendwas mit Matrix[x][y]
    return true;
}
}
```

8.5.2 Die continue-Anweisung

Darf nur in Schleifen auftreten. Beendet die aktuelle Iteration und fährt mit der Auswertung der Schleifenbedingung fort.

`continue` **mit** Label beendet die aktuelle Iteration der markierten Schleife.

`continue` **ohne** Label beendet die aktuelle Iteration der direkt umgebenden Schleife.

Beispiel:

```
while (! stream.eof())
{ token = stream.next();
  if (token.equals("skip"))
    continue;
  // Verarbeite token
}
```

8.5.3 Die return-Anweisung

Beendet die Ausführung einer Methode und kehrt zum Aufrufer zurück.

Liefert die Methode kein Ergebnis, ist die Form

```
return;
```

Liefert die Methode ein Ergebnis, ist die Form

```
return expression;
```

wobei `expression` einen Typ haben muß, der an den Typ des Methodenergebnisses zugewiesen werden kann.

Besonderheit:

`return` kann auch zum Verlassen von Konstruktoren und Initialisierungsblöcken verwendet werden.

9 Ausnahmen

Ausnahmen (*Exceptions*) bieten die Möglichkeit, Fehlersituationen zu behandeln, ohne den Code unleserlich zu machen.

Java bietet zwei Arten von Ausnahmen:

- **ungeprüfte:** Ausnahmen der Klassen `Error` und `RuntimeException`, für die der Compiler keine Überprüfung auf korrekte Behandlung durchführt.
- **geprüfte:** Benutzerdefinierte (und vordefinierte) Ausnahmen. Man deklariert, welche Ausnahmen eine Methode erzeugen kann (`throws`-Klausel). Compiler überprüft, daß Methoden keine unpassenden Ausnahmen erzeugen.

Eine Ausnahme wird bei unerwarteten Umständen ausgelöst (*throw*). Sie wird abgefangen (*catch*) von einem dafür zuständigen Konstrukt (weiter vorne in der Methodenaufaufrufkette). Falls ein solches nicht existiert, übernimmt ein *Default Exception Handler*.

9.1 Definieren von Ausnahmetypen

Exceptions in Java sind Objekte. Alle benutzerdefinierten (und vordefinierten) Exceptions erweitern die Klasse `Exception`.

Das `Attributed`-Interface (Seite 88) erweitern wir um eine Methode `replaceValue(name, newval)`.

Wenn ein Attribut mit Namen `name` nicht existiert, soll eine Ausnahme `NoSuchAttrException` ausgelöst werden:

```
public class NoSuchAttrException
extends Exception
{ public String attrName;
  public String newVal;
  NoSuchAttrException(String name,
                      Object val)
  { super("No such attribute " + name);
    attrName = name;
    newVal = val;
  }
}
```

Die Ausnahme `NoSuchAttrException` ist Problem-spezifisch und besser geeignet, die Fehlersituation zu beschreiben als allgemeine Ausnahmetypen.

9.2 Auslösen von Ausnahmen

Exceptions werden durch die `throw`-Anweisung ausgelöst, die die Programmausführung sofort abbricht.

Für unsere Erweiterung des `Attributed`-Interfaces in der neuen Methode

```
public void replaceValue(String name,
                        Object newval)
    throws NoSuchAttrException
{ Attr attr = find(name);
  if (attr == null)
    throw new NoSuchAttrException
        (name, newval);
  attr.valueOf(newValue);
}
```

Die `throws`-Klausel gibt an, welche Exceptions von einer Methode ausgelöst werden können:

```
type method(parameters)
    throws excpt1, excpt2, excpt2
...
```

Alle Exception-Objekte aus Unterklassen von `excpt1`, `excpt2` und `excpt3` können von `method` ausgelöst werden.

Man könnte auch eine gemeinsame Oberklasse von `excpt1`, `excpt2` und `excpt3` verwenden, verschleiert damit aber möglicherweise nützliche Information.

Die **ungeprüften** Ausnahmen

- `RuntimeException`

- `Error`

werden nicht in der `throws`-Klausel angegeben. Sie können im Prinzip von jeder Methode ausgelöst werden. `throws`-Klauseln dafür würden den Code unleserlich machen.

9.3 Abfangen von Ausnahmen

Ausnahmen werden abgefangen, indem man Programmcode in `try`-Klauseln einschließt.

```
try block
catch (exception_type id) block
catch (exception_type id) block
...
finally block
```

Ausführung

- Werte den `try`-Block aus.
- Gibt es eine Exception,: bestimme passende `catch`-Klausel (Suche von oben nach unten).
 - falls passende `catch`-Klausel existiert: führe deren Block und anschließend den `finally`-Block aus.
 - falls passende `catch`-Klausel **nicht** existiert: führe `finally`-Block aus. Gib Exception an umgebende `try`-Klauseln bzw. an aufrufenden Code weiter.
- Gibt es keine Exception, führe `finally`-Block aus.

Die Regeln sind eigentlich etwas komplizierter, da die `catch`- und `finally`-Blöcke ihrerseits Ausnahmen auslösen können.

Unser Beispiel:

```
try
{ attributedObj.replaceValue("Alter", 32);
```

```
}  
catch (NoSuchAttrException e)  
// kann nicht vorkommen.  
// falls es doch vorkommt,  
// korrigieren wir es:  
{ Attr a = new Attr(e.attrName,  
                    e.newVal);  
  attributedObj.add(a);  
}
```

`finally` ist bisweilen auch ohne Ausnahmebehandlung nützlich:

```
try  
{ input = new Stream(file);  
  while (!input.eof())  
    if (input.next() == word)  
      return true;  
  return false; // nicht gefunden  
}  
finally  
{ if (input != null)  
  input.close();  
}
```

Der `finally`-Block wird immer mit einem bestimmten Grund betreten:

- `try`-Block normal verlassen
- `try`-Block durch Ablauf-Anweisung verlassen, z.B. `return`
- `try`-Block durch Exception verlassen.

Wird `finally` normal verlassen, bleibt dieser Grund für das gesamte `try` bestehen.

Wird `finally` wegen `Exception` oder `return` verlassen, bestimmt dies den “Grund” der gesamten `try`-Anweisung.

Beispiel:

```
try
{
    ....
    return 1;
}
finally
{
    return 2;
}
```

Gibt 2 zurück. Das wäre sogar so, wenn der `try`-Block eine Ausnahme ausgelöst hätte.

Noch ein Beispiel:

```
class TestExcpt extends Exception
{
    TestExcpt() {}
    TestExcpt(String s) {super(s);}
}

public class Test
{
    public static void main(String[] a)
    {
        for (int i=0; i < a.length; i++)
        {
            try { werfer(Integer.parseInt(a[i]));
                System.out.println("no exception");
            }
            catch (Exception e)
            {
                System.out.println (e.getClass() + "\n\t"
                                    + e.getMessage());
            }
        }
    }

    static int werfer(int i) throws TestExcpt
    {
        switch (i)
        {
            case 0: return i/i;
            case 1: return (new int[-9]).length;
            case 2: throw new TestExcpt("Test");
        }
    }
}
```

```
        return 0;
    }
}
```

Dieses Beispiel erzeugt beim Aufruf

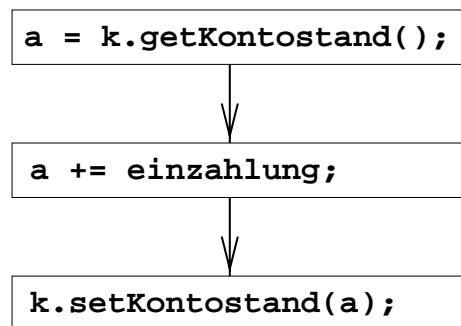
```
java Test 0 1 2 3
```

die Ausgabe

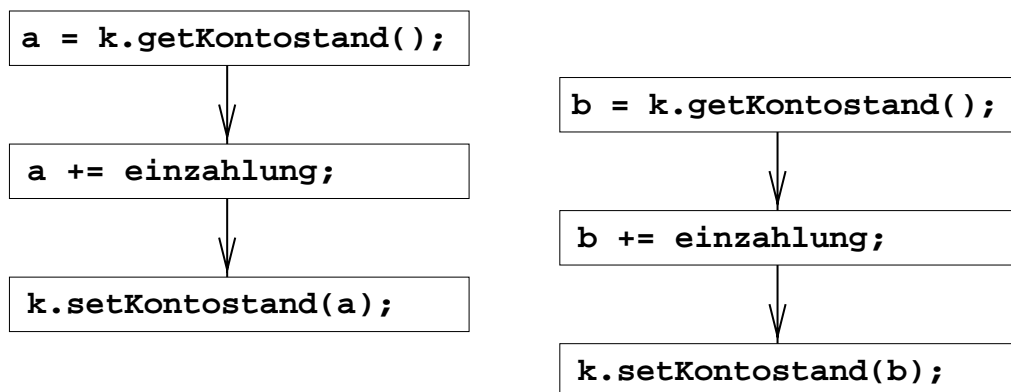
```
class java.lang.ArithmeticException
    / by zero
class java.lang.NegativeArraySizeException
    null
class TestExcpt
    Test
no exception
```

10 Parallelausführung

In den meisten Programmiersprachen arbeitet man mit einzelnen sequentiellen Ausführungssträngen (*single threaded*).



Realistischer ist oft die Modellierung durch parallele Ausführungsstränge (*multi threaded*).



Arbeiten parallele Ausführungsstränge auf gemeinsamen Daten, entstehen zufällige Ergebnisse (*race hazards*), wenn die Threads nicht bezüglich dieser Daten synchronisiert werden.

Java bietet die Mechanismen zur Programmierung paralleler Ausführungsstränge:

- Erzeugen von Threads
- Starten, Suspendieren und Stoppen von Threads

- Synchronisation bezüglich gemeinsam benutzter Objekte
- Kommunikation zwischen Threads
- Beeinflussung des Scheduling-Prozesses

10.1 Starten von Threads

Die Klasse `Thread` steht in der Bibliothek `java.lang` zur Verfügung.

Es gibt zwei Arten, Threads zu definieren und zu starten:

1. Möglichkeit

Definiere eine Unterklasse von `Thread`. Überschreibe deren Methode `run`. Dann werden Objekte dieser Unterklasse erzeugt und gestartet (Methode `start`).

Beispiel:

```
class PrimThread extends Thread
{
    long minPrim;
    PrimThread(long minPrim)
    {
        this.minPrim = minPrim;
    }
    public void run()
    {
        // Berechne Primzahlen
        // größer als minPrim
        . . .
    }
}
```

Starten durch:

```
PrimThread p = new PrimThread(143);
p.start();
```

2. Möglichkeit

Definiere eine Klasse, die das Interface `Runnable` implementiert. Erzeuge ein Objekt dieser Klasse, übergebe es als Argument der Thread-Erzeugung und starte den erzeugten Thread.

Beispiel:

```
class PrimRun implements Runnable
{ long minPrim;
  PrimRun(long minPrim)
  { this.minPrim = minPrim;
  }
  public void run()
  { // Berechne Primzahlen
    // größer als minPrim
    . . .
  }
}
```

Starten durch:

```
PrimRun p = new PrimRun(143);
new Thread(p).start();
```

Beispiel:

```
class Uthread extends Thread
{ public Uthread(String str)
  { super(str); // Thread hat einen Namen
  }
  public void run()
  { for (int i = 0; i < 5; i++)
    { System.out.println(i+" "+getName());
      try
      { sleep((int)(Math.random()*1000));
      }
    }
  }
}
```

```
        }
        catch (InterruptedException e) {}
    }
    System.out.println("1998: "+getName());
}
}

class Urlaub
{ public static void main (String[] args)
  { new Uthread("Sauerland").start();
    new Uthread("Malediven").start();
  }
}
```

Könnte z.B. folgende Ausgabe erzeugen:

```
0 Sauerland
0 Malediven
1 Sauerland
2 Sauerland
3 Sauerland
1 Malediven
4 Sauerland
1998: Sauerland
2 Malediven
3 Malediven
4 Malediven
1998: Malediven
```

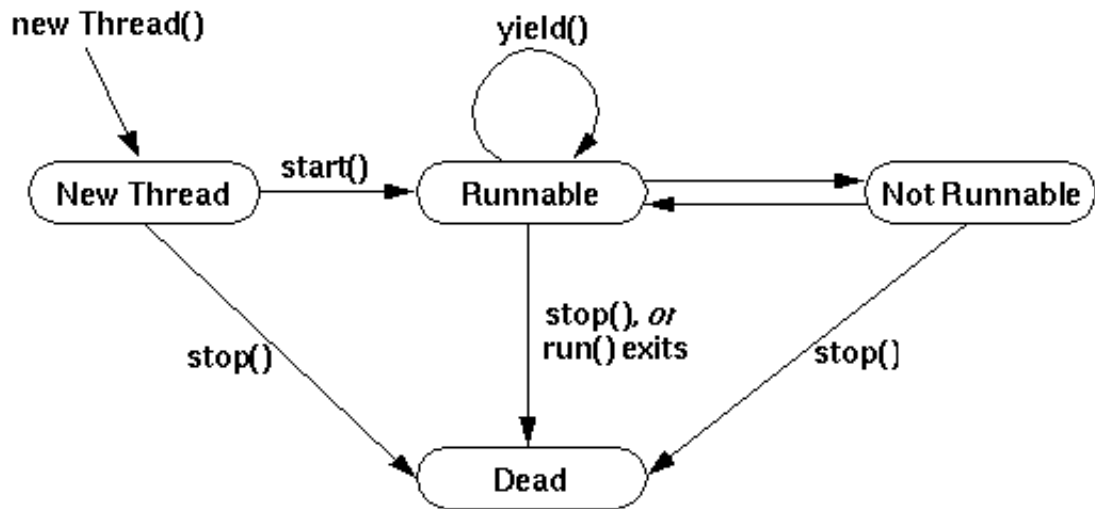
Wichtiges Anwendungsfeld für Threads sind auch Applets. Sie brauchen Threads, um die Anzeige regelmäßig auffrischen zu können ohne andere Abläufe zu behindern.

Beispiel: Digitaluhr

```
import java.awt.Graphics;
import java.util.Date;
```

```
public class Clock
extends java.applet.Applet
implements Runnable
{ Thread digital;
  public void start()
  { if (digital == null)
    { digital = new Thread(this, "Clock");
      digital.start();
    }
  }
  public void run()
  { while (digital != null)
    { repaint();
      try { Thread.sleep(1000);}
      catch (InterruptedException e){}
    }
  }
  public void paint(Graphics g)
  { g.drawString
    (new Date().toString(), 5, 10);
  }
  public void stop()
  { digital = null;
  }
}
```

10.2 Zustand eines Threads



Für den Übergang zwischen `Runnable` und `Not Runnable` gibt es vier Gründe:

- Aufruf der `sleep()`-Methode. `Runnable` wieder nach Ablauf der angegebenen Zeit.
- Aufruf der `suspend()`-Methode. `Runnable` wieder nach Aufruf der `resume()`-Methode.
- Der Thread benutzt `wait()`, um auf eine Bedingungsvariable zu warten. Weiter geht es, wenn Besitzer der Bedingungsvariablen entweder `notify()` oder `notifyAll()` aufruft (siehe Seite 140).
- Der Thread blockiert wegen Ein-/Ausgabe. Weiter geht es, wenn die Ein-/Ausgabe-Aktion abgeschlossen ist.

10.3 Synchronisation von Threads über Daten

Synchronisation von Threads dient der Sicherstellung des gegenseitigen Ausschlusses beim Zugriff auf gemeinsam benutzte Daten.

Java bietet zwei Konstrukte

- `synchronized`-Methoden
- `synchronized`-Anweisungen

10.3.1 Synchronisierte Methoden

Wird eine als `synchronized` markierte Methode für ein Objekt aufgerufen, so gilt dieses Objekt als **gesperrt** (*locked*).

Alle anderen Threads, die ebenfalls eine synchronisierte Methode für dieses Objekt aufrufen, werden blockiert, bis diese **Sperre** aufgehoben ist.

So läßt sich gegenseitiger Ausschluß beim Zugriff auf Daten sicherstellen.

Die Java-Synchronisation beruht auf **Monitoren** (Hoare, 1974) (siehe Seite 140). Es gibt allerdings keine expliziten *lock*- und *unlock*-Operationen.

Beispiel:

```
class Konto
{ private double Kontostand;
  public Konto(double anf)
  { Kontostand = anf;
  }
  public synchronized
    double getKontostand()
  { return Kontostand;
  }
  public synchronized
    void setKontostand(double d)
  { Kontostand = d;
  }
  public synchronized
    void einzahlen(double b)
  { double a = getKontostand();
    a += b;
    setKontostand(a);
  }
}
```

Synchronisierte Klassenmethoden

synchronized-Markierung geht auch bei Klassenmethoden :

- Aufruf einer synchronisierten Klassenmethode bewirkt gegenseitigen Ausschluß aller anderen synchronisierten Klassenmethoden (*Klassen-Sperre*).
- Dies hat keinen Einfluß auf den Aufruf von synchronisierten Instanzmethoden (*Objekt-Sperre*).

Überschreiben von synchronisierten Methoden

Eine Methode, die eine synchronisierte Methode überschreibt, kann entweder synchronisiert oder nicht synchronisiert sein.

Die Eigenschaft der überschriebenen Methode, Objekte zu sperren, geht durch die nicht synchronisierte überschreibende Methode nicht verloren.

```
class Ober
{ synchronized void method()
  { ...
  }
  ...
}
class Unter extends Ober
{ void method()
  { super.method(); // jetzt sperren
  ...
  }
  ...
}
```

10.3.2 Synchronisierte Anweisungen

Synchronisierte Anweisungen dienen dazu, bezüglich Objekten zu synchronisieren auch wenn keine synchronisierten Methoden vorhanden sind.

Form:

```
synchronized(expr)
  statement
```

Bedeutung:

Wenn der Thread die Sperre für das Objekt `expr` erhält, kann die Anweisung `statement` ausgeführt werden.

Beispiel:

```
// numbs soll nach der folgenden
// Schleife Absolutwerte enthalten
synchronized (numbs)
{ for (int i = 0; i < arr.length; i++)
    if (numbs[i] < 0)
        numbs[i] = -numbs[i];
}
```

Die Sperren, die zur Realisierung synchronisierter Anweisungen verwendet werden, sind die gleichen wie für synchronisierte Methoden.

Die beiden Synchronisations-Schreibweisen von Java können also zusammenarbeiten.

10.4 Synchronisation von Threads mit `wait` und `notify`

Die bisher betrachteten Objekt-Sperren stellen den gegenseitigen Ausschluß beim Zugriff auf gemeinsam benutzte Daten sicher.

Aktiv zusammenarbeitende Threads müssen zusätzlich über Eigenschaften gemeinsam benutzter Ressourcen informiert sein.

Typisches Beispiel: Produzent/Konsument-Kommunikation

```
class Produzent extends Thread
```

```
{ private Puffer puffer;

    public Produzent(Puffer p)
    { puffer = p;
    }

    public void run()
    { for (int i = 0; i < 5; i++)
      { puffer.put(i);
        System.out.println
          ("Produzent gibt: " + i);
        try
        { sleep((int)(Math.random() * 1000));
        }
        catch (InterruptedException e) {}
      }
    }
}
```

```
class Konsument extends Thread
{ private Puffer puffer;

    public Konsument(Puffer p)
    { puffer = p;
    }

    public void run()
    { int wert = 0;
      for (int i = 0; i < 5; i++)
      { wert = puffer.get();
        System.out.println
          ("Konsument nimmt " + wert);
      }
    }
}
```

Das Hauptprogramm, das Produzent und Konsument startet:

```
class P_K_Test
{ public static void main(String[] args)
  { Puffer p = new Puffer();
    new Produzent(p).start();
    new Konsument(p).start();
  }
}
```

Annahme: Der benutzte Puffer kann genau eine Zahl aufnehmen:

Wenn Produzent und Konsument sich nicht über Eigenschaften der gemeinsam benutzten Ressource vom Typ `Puffer` verständigen, wird das Ergebnis falsch:

- Ist der Produzent schneller, verpasst der Konsument Zahlen.
- Ist der Konsument schneller, bekommt er manche Zahlen mehrfach.

Die Synchronisation zwischen Produzent und Konsument geschieht in Java über **Monitore**.

- Ein Monitor existiert für jedes Objekt (*Bedingungsvariable*), das `synchronized` Methoden hat.
- Nur ein Thread kann zu einer Zeit den Monitor betreten.
- Andere Threads, die in dieser Zeit synchronisierte Methoden für das Objekt aufrufen, müssen warten.

Die `wait()`-Methode

Ein Thread im Monitor kann `wait()` aufrufen, um den Monitor freizugeben, während er auf eine bestimmte Bedingung wartet.

Die `notify()`- und `notifyAll()`-Methoden

Ein Thread im Monitor kann `notifyAll()` oder `notify()` aufrufen, um andere Threads, die wegen der gleichen Bedingungsvariable warten, wieder ausführbereit zu machen.

`notifyAll` macht alle wartenden Threads ausführbereit.

`notify` wählt aus den wartenden Threads einen aus, der ausführbereit wird.

Der Puffer für das Produzent/Konsument-Schema

```
class Puffer {
    // Pufferelement zwischen Produzent
    // und Konsument
    private int inhalt;
    private boolean gefüllt = false;

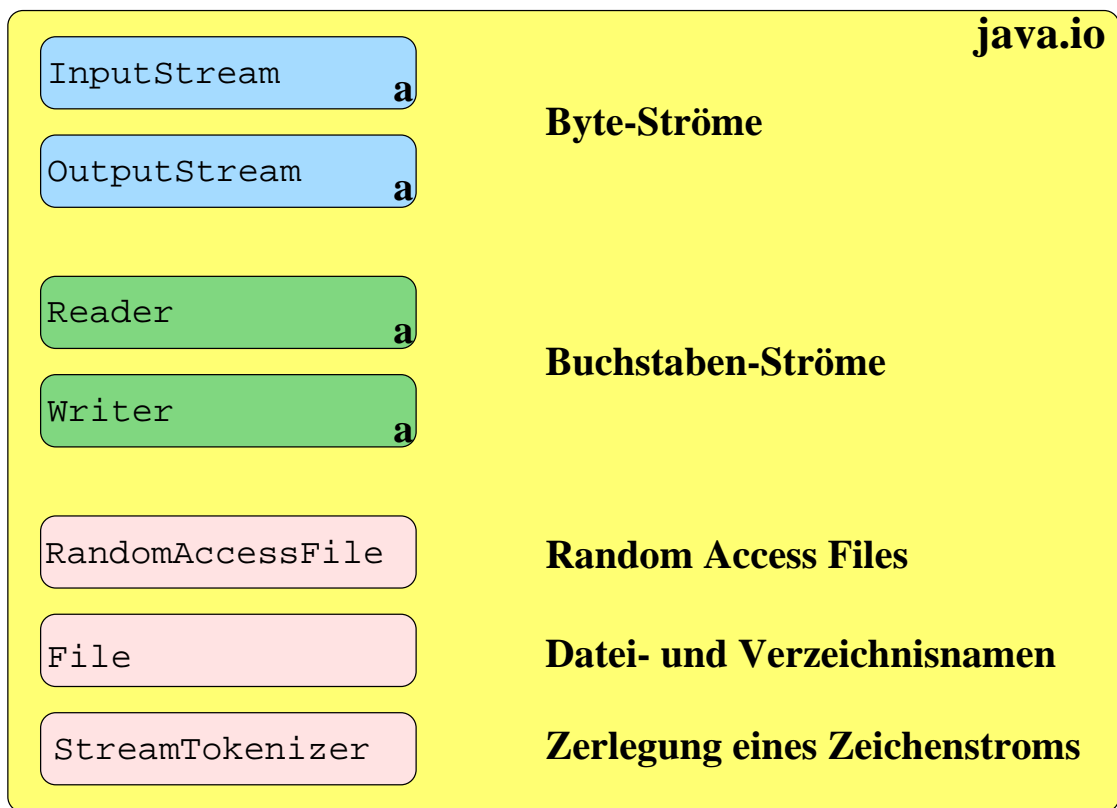
    public synchronized int get()
    { while (gefüllt == false)
      { try {wait();}
        catch (InterruptedException e) {}
      }
      gefüllt = false;
      notifyAll();
      return inhalt;
    }

    public synchronized void put(int wert)
    { while (gefüllt == true)
      { try { wait();}
        catch (InterruptedException e) {}
      }
      inhalt = wert;
      gefüllt = true;
      notifyAll();
    }
}
```

11 Ein-/Ausgabe

Ein- und Ausgabe ist im Paket `java.io` implementiert. Die gliedert sich in die Hauptbereiche

- Ein-/Ausgabe mit Byte-Strömen
- Ein-/Ausgabe mit Buchstaben-Strömen
- Random Access Dateien
- File-Namen-Behandlung
- Zerlegen von Strömen in Symbole (Scanning)



a: abstrakte Klasse

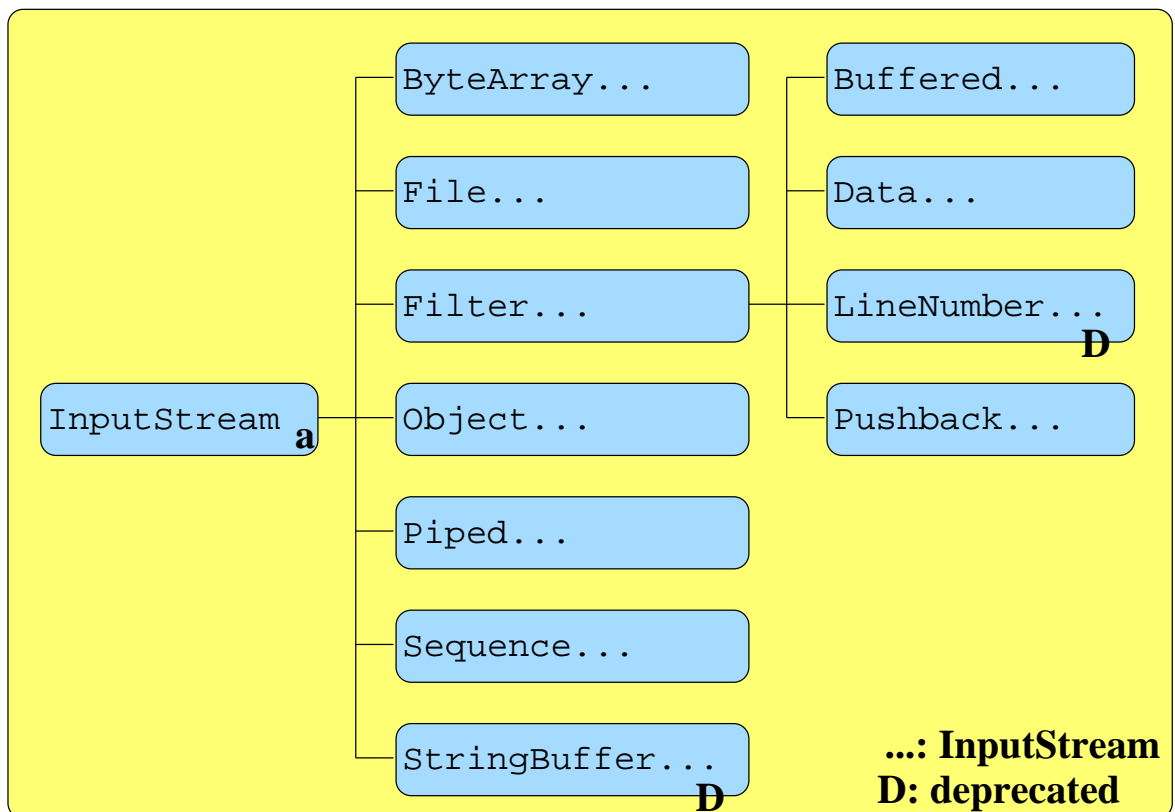
Die Klassen aus `java.io` benutzt man meist nicht einzeln sondern in Kombination.

11.1 Byte-Ströme

Byte-Ströme dienen zur Speicherung binärer Daten

11.1.1 Binäre Eingabe-Ströme

Byte-Strom-Eingabe wird von der abstrakten Klasse `InputStream` realisiert:



Beispiel: Zähle die Zeichen in einer Datei

```
import java.io.*;

class CountChars
{ public static void main(String argv[])
  throws IOException
```



```
{ InputStream in;
  if (argv.length != 0)
    in = new FileInputStream(argv[0]);
  else in = System.in;
  int ch;
  int total=0;
  while ((ch = in.read()) != -1)
    total++;
  System.out.println("Number of chars: "
                    + total);
}
```

Neben dem oben verwendeten `FileInputStream` existieren:

- `ByteArrayInputStream`:
Daten aus einem Byte-Array im Speicher lesen.
- `ObjectInputStream`:
Teil des *Object Serialization APIs* (neu in Java1.1) zur Ausgabe von Objekten (siehe Seite 156).
- `PipedInputStream`:
implementiert die Eingabe-Seite einer Pipe.
- `SequenceInputStream`:
konkateniert mehrere Eingabeströme zu einem (siehe Seite 144).
- `StringBufferInputStream`:
Lesen aus `StringBuffer`-Objekten (veraltet, daher: deprecated-Meldung).
- `FilterInputStream`: Diese Klassen erlauben die Filterung gelesener Daten (siehe Seite 146).

Folgen von Eingabeströmen

`SequenceInputStream` wird verwendet um ein Folge von Eingabeströmen zu verarbeiten:

```

import java.io.*;
class Concatenate
{ public static void main(String[] a)
  { ListOfFiles mylist = new ListOfFiles(a);
    try
    { SequenceInputStream s =
      new SequenceInputStream(mylist);
      int c;
      while ((c = s.read()) != -1)
        System.out.write(c);

      s.close();
    }
    catch (IOException e)
    { System.err.println
      ("Concatenate: " + e);
    }
  }
}

```

Hier ist die notwendige Enumeration:

```

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

class ListOfFiles implements Enumeration
{ String[] listOfFiles;
  int current = 0;
  ListOfFiles(String[] listOfFiles)
  { this.listOfFiles = listOfFiles;
  }
  public boolean hasMoreElements()
  { return current < listOfFiles.length;
  }
  public Object nextElement()
  { InputStream is = null;
    if (!hasMoreElements())
      throw new NoSuchElementException();
    else
    { try

```

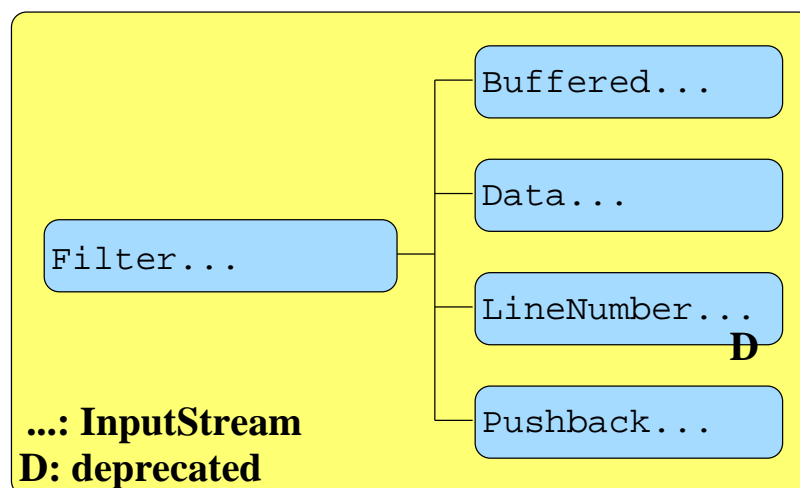
```

        { String next = listOfFiles[current++];
          is = new FileInputStream(next);
        }
        catch (FileNotFoundException e) {}
      }
      return is;
    }
  }
}

```

Gefilterte Binär-Eingabe

Filterströme werden mit anderen Eingabeströmen verbunden, um die eingelesenen Daten weiterzubearbeiten.



Neben der Verwendung dieser Filter-Strom-Klassen aus der Bibliothek besteht die Möglichkeit, eigene Filterklassen zu schreiben.

Benutzung am Beispiel von `DataInputStream`:

```

import java.io.*;
class CountLines

```

```
{
    public static void main(String a[])
        throws IOException
    {
        InputStream in;
        if (a.length != 0)
            in = new FileInputStream(a[0]);
        else in = System.in;

        DataInputStream dis =
            new DataInputStream(in);
        String inp;
        int total = 0;
        while ((inp = dis.readLine()) != null)
            total++;
        System.out.println("Number of lines: "
                           + total);
    }
}
```

DataInputStream liefert Einlesemethoden für alle gängigen Datentypen, u.a.:

Method Index

```
public final byte readByte();
public final char readChar();
public final double readDouble();
public final float readFloat();
public final int readInt();
public final String readLine();
public final long readLong();
public final short readShort();
```

Gepufferte Binär-Eingabe

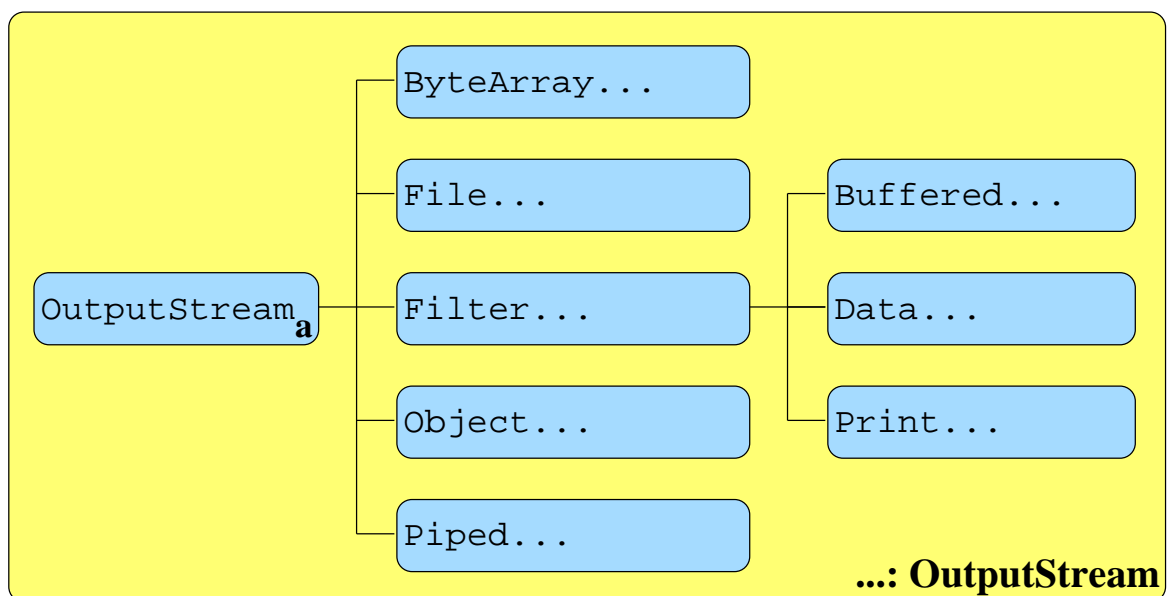
Die Klasse `BufferedInputStream` steigert die Effizienz der Eingabe, indem größere Datenmenge in einem internen Puffer zwischengespeichert werden.

Beispiel:

```
public class InFile extends DataInputStream
{ public InFile(String filename)
  throws FileNotFoundException
  { super(
    new BufferedInputStream(
      new FileInputStream(filename)));
  }
}
```

11.1.2 Binäre Ausgabe-Ströme

Byte-Strom-Ausgabe wird von der abstrakten Klasse `OutputStream` realisiert:



Beispiel:

```
import java.io.*;
class BinIO
{ public static void main(String[] args)
  { try
    { FileInputStream fis =
      new FileInputStream("try.txt");
      FileOutputStream fos =
        new FileOutputStream("try_cp.txt");
      int c;
      while ((c = fis.read()) != -1)
      { fos.write(c);
      }
      fis.close();
      fos.close();
    }
    catch (IOException e)
    { System.err.println("BinIO: " + e);
    }
  }
}
```

Ausgabe textueller Darstellung der Grundtypen

Die Klasse `PrintStream` liefert Methoden zur Ausgabe der textuellen Darstellung der Java-Grundtypen.

- `print`: überladen für alle Grundtypen
- `println`: dto. mit Ausgabe eines abschließenden Zeilenendes.

Da sie eher zur Zeichenausgabe (siehe Seite 152) gehört, wird die Klasse nicht mehr benutzt. Statt dessen kommt die Klasse `PrintWriter` (siehe Seite 152) zum Einsatz.

Einzigste Bedeutung:

Die beiden Standard-Ausgabedateien `System.out` und `System.err` sind vom Typ `PrintStream`.

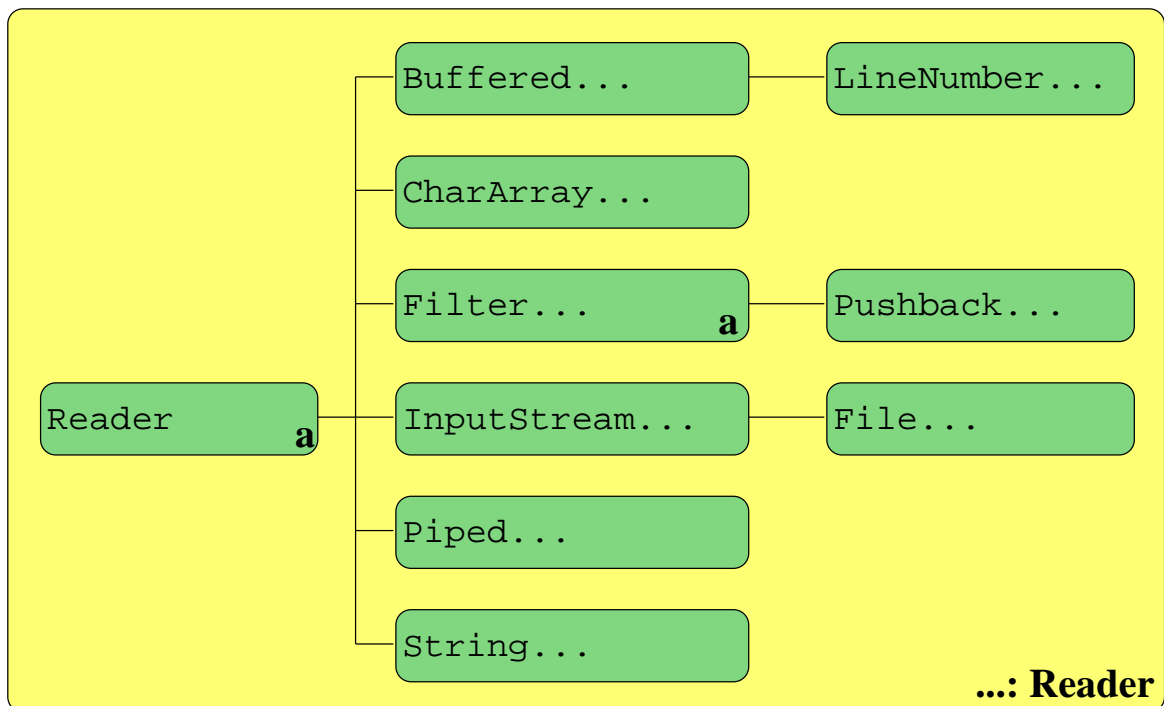
11.2 Zeichen-Ströme

Zeichen-Ströme dienen zur Speicherung von Texten. Sie wurden in Java1.1 eingeführt.

Im Gegensatz zu den Byte-Strömen, die bis Java1.0 auch zur Zeichenausgabe verwendet wurden, garantieren die Zeichen-Strom-Klassen die korrekte Umsetzung von Unicode gemäß der lokalen Gegebenheiten.

11.2.1 Zeichen-Eingabe-Ströme

Zeichen-Strom-Eingabe wird von der abstrakten Klasse Reader realisiert:



Anwendung: Gepufferte Texteingabe

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

```
public class ReadLines
{ public static void main (String a[])
  { BufferedReader din =
    new BufferedReader(
      new InputStreamReader(System.in));
    String line;
    do { try
      { line = din.readLine();
      }
      catch(IOException ioe)
      { break;
      }
      if (line == null) // end of stream
        break;
      System.out.println(line);
    }
    while (true);
  }
}
```

Anwendung: Texteingabe aus Strings

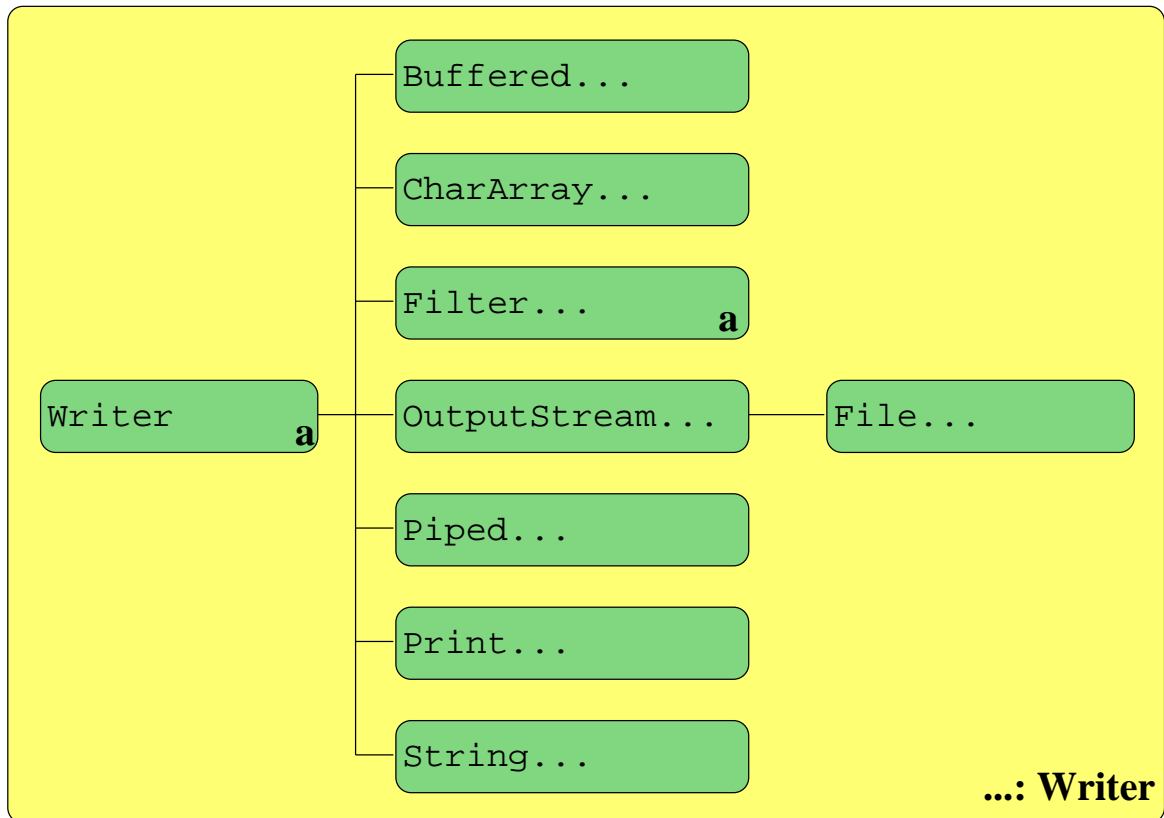
Die Klasse `StringReader` bietet die Standard-Funktionalität der Reader-Klassen, wobei die Lesquelle ein `String` ist.

Beispiel:

```
import java.io.*;
class StrRead
{ public static void main(String[] a)
  throws IOException
  { StringReader inp =
    new StringReader(a[0]);
    int c;
    while((c = inp.read()) != -1)
      System.out.println((char)c);
  }
}
```


11.2.2 Zeichen-Ausgabe-Ströme

Zeichen-Strom-Ausgabe wird von der abstrakten Klasse `Writer` realisiert:



Gepufferte Zeichenausgabe für die Grundtypen

Die Klasse `PrintWriter` bietet Methoden für die Zeichenausgabe der Java-Grundtypen. Sie ersetzt in Java1.1 die Klasse `PrintStream` (siehe Seite 149).

Beispiel:

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

```
class SquareList
{ public static void main(String[] a)
  throws IOException
  { PrintWriter out =
    new PrintWriter(
      new BufferedWriter(
        new FileWriter("square.list")));
    for (int i = 0;
        i < Integer.parseInt(a[0]);
        i++)
    { out.print(i);
      out.print(": ");
      out.println(Math.sqrt(i));
    }
    out.close();
  }
}
```

11.3 Random Access Dateien

Ziel der `RandomAccessFile`-Klasse ist die freie Positionierung von Schreib-/Lese-Zeigern an beliebigen Datei-Positionen.

Beispiel:

```
import java.io.*;
class RandAcc
{ public static void main(String[] a)
  throws IOException
  { RandomAccessFile rf;
    rf = new RandomAccessFile(
      "try.dat", "rw");
    rf.writeChars("Jva");
    rf.close();
    rf = new RandomAccessFile(
      "try.dat", "rw");
    rf.seek(2); // Position 2
    rf.writeChars("ava");
  }
}
```

```
rf.close();

BufferedReader din =
    new BufferedReader(
        new FileReader("try.dat"));
// schreibe "Java":
System.out.println(din.readLine());
}
}
```

11.4 Datei- und Verzeichnis-Namen

Die Klasse `File` bietet nützliche Methoden, um Datei- und Verzeichnisnamen zu manipulieren.

Außerdem erlauben viele der Konstruktoren der IO-Klassen ein `File`-Argument:

```
File ipf = new File("input.dat");
FileInputStream fis =
    new FileInputStream(ipf);
```

Einige nützliche Methoden

Method Index

```
boolean delete()
boolean exists()
boolean isDirectory()
boolean isFile()
long length()
String[] list()
```

```
String [] list(FilenameFilter) (z.B.: Seite 63)

boolean mkdir()

boolean renameTo(File)
```

Beispiel: Löschen von Dateien

```
import java.io.*;
public class Delete // Lösche eine Datei
{ public static void main(String[] args)
  { try { delete(args[0]);}
    catch (IllegalArgumentException e)
    { System.err.println(
      e.getMessage() + ": " + args[0]);
    }
  }
  public static void delete(String name)
  { File f = new File(name);
    if (!f.exists())
      fail("No such file or directory");
    if (!f.canWrite())
      fail("Write protected");
    if (f.isDirectory())
    { String[] files = f.list();
      if (files.length > 0)
        fail("Directory not empty");
    }
    if (!f.delete()) // jetzt endlich
      fail("Deletion failed");
  }
  protected static void fail(String msg)
  throws IllegalArgumentException
  { throw new IllegalArgumentException(msg);
  }
}
```

11.5 Zerlegung von Strömen in Einzelsymbole

Mit `StreamTokenizer` bietet Java eine Klasse für einfache Zerlegung von Eingabeströmen in Einzelsymbole.

Der `StreamTokenizer` erkennt 4 einfache Symbolklassen

- `TT_WORD`
- `TT_NUMBER`
- `TT_EOL`
- `TT_EOF`

und stellt deren Werte in öffentlichen Attributen zur Verfügung.

Kleines Beispiel:

```
static double sumStream(InputStream in)
throws IOException
{ StreamTokenizer nums =
    new StreamTokenizer(in);
  double result = 0.0;
  while (nums.nextToken() ==
    StreamTokenizer.TT_NUMBER)
    result += nums.nval;
  return result;
}
```

11.6 Ausgabe von Objekten (Serialisierung)

`Serializable` ist ein ("Marker"-) Interface in `java.io`, das dazu benutzt wird, Klassen zu kennzeichnen, von denen Objekte in Binärdateien abgelegt werden sollen (neu in Java1.1).

Serialisierung ist wichtig in Client/Server Anwendungen, für Cut-and-Paste, für Software-Komponenten (*Beans*) und die Speicherung bereits initialisierter Applets.

Beispiel: Serialisierung von baumstrukturierten Objekten**Eine Klasse für Suchanfragen**

Markiert als Serializable

```
import java.io.*;

abstract class SuchAnfrage
implements Serializable
{abstract public void print();
}
```

Ein einzelner Suchbegriff

```
import java.io.*;
public class Begriff extends SuchAnfrage
{ String begr;
  Begriff(String b)
  { begr = b;
  }
  public void print()
  { System.out.print(begr + " ");
  }
}
```

Und-Verknüpfung von Suchbegriffen

```
import java.io.*;
public class Und extends SuchAnfrage
{ SuchAnfrage sl;
  SuchAnfrage sr;
  public Und(SuchAnfrage sl,
            SuchAnfrage sr)
  { this.sl = sl;
    this.sr = sr;
  }
  public void print()
  { System.out.print("(" );
```

```
        sl.print();
        System.out.print("UND ");
        sr.print();
        System.out.print(" ");
    }
}
```

Oder-Verknüpfung von Suchbegriffen

```
import java.io.*;
public class Oder extends SuchAnfrage
{
    SuchAnfrage sl;
    SuchAnfrage sr;
    public Oder(SuchAnfrage sl,
                SuchAnfrage sr)
    {
        this.sl = sl;
        this.sr = sr;
    }
    public void print()
    {
        System.out.print("(" );
        sl.print();
        System.out.print("ODER ");
        sr.print();
        System.out.print(") ");
    }
}
```

Negation von Suchbegriffen

```
import java.io.*;
public class Nicht extends SuchAnfrage
{
    SuchAnfrage s;
    Nicht(SuchAnfrage s)
    {
        this.s = s;
    }
    public void print()
    {
        System.out.print("NICHT ");
        s.print();
    }
}
```

Das Hauptprogramm: Serialisierung von Suchbegriffen

Zunächst wird eine Suchanfrage konstruiert:

```
(Java UND NICHT (C++ ODER Smalltalk))
```

```
import java.io.*;

public class Main
{ public static void main(String [] a)
  throws IOException,
    ClassNotFoundException
{ SuchAnfrage s;
  s = new Und(
    new Begriff("Java"),
    new Nicht(
      new Oder(
        new Begriff("C++"),
        new Begriff("Smalltalk")
      )
    )
  );

  // Objekte rausschreiben
  ObjectOutputStream out =
    new ObjectOutputStream(
      new FileOutputStream("such.dat"));
  out.writeObject(s);
  out.close();

  // Objekte einlesen
  ObjectInputStream in =
    new ObjectInputStream(
      new FileInputStream("such.dat"));
```



```
SuchAnfrage wiederda =  
    (SuchAnfrage) in.readObject();  
wiederda.print();  
System.out.println();  
}  
}
```

12 Grafische Benutzungsschnittstellen

Das Paket `java.awt` (*abstract windowing toolkit*) dient zur Realisierung grafischer Benutzungsschnittstellen:

- Applets: Grafik-basierte Programme, die unter Kontrolle eines Browsers im Sandkasten-Modell ablaufen.
- Applikationen: Grafik-basierte “normale” Anwendungsprogramme.

Das `awt`-Paket wurde für Java1.1 stark revidiert.

Vor allem: Ersetzen der nicht-objektorientierten Ereignisbehandlung.

Entwicklung einer grafischen Benutzungsschnittstelle erfordert 4 Schritte:

- Erzeugen der **Grafik-Komponenten**. In Java: Konstruktoraufruf:

```
Button quit = new Button("quit");
```

- Hinzufügen der Grafikkomponente zu einem **Container** (z.B. Rahmen, Dialog, Zeichenfläche):

```
myContainer.add(quit);
```

- Anordnen der Komponenten in ihren Containern. In Java zuständig: **Layout Manager**.
- Behandlung der **Ereignisse**, die von den Komponenten erzeugt werden (*event handling*). Sehr verschiedene Modelle in Java1.0 und Java1.1.

12.1 Das Ereignis-Modell des AWT (ab JDK1.1)

Ereignisse werden von *Ereignisquellen* erzeugt. Ein oder mehrere **Listener** melden sich an, um über die Ereignisse an einer Quelle informiert zu werden.

Das Modell heißt **Delegation** :

die Ereignisbehandlung wird an irgendein Objekt delegiert, das das zum Ereignis passende Interface implementiert.

Zur Ereignisbehandlung brauchen wir folgendes

1. Eine Klasse für die Ereignisbehandlung. Sie muß das Interface für das zugehörige Ereignis implementieren, z.B.:

```
public class MyEvtHdl
    implements ActionListener
{ ...
```

2. Anmelden eines Listeners bei einer Ereignisquelle, z.B.:

```
quitknopf.addActionListener(
    new MyEvtHdl());
```

3. Implementierung der versprochenen Event-Handling-Methoden, z.B.:

```
public void actionPerformed(
   (ActionEvent e)
{ // irgendwie auf das
  // Ereignis reagieren
```

Beispiel: Ein piepsender Knopf

Hier ist der `ActionListener` selbst Besitzer des Knopfes. Das muß natürlich nicht so sein.

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

public class Beeper
    extends Applet
    implements ActionListener
{ Button button;
  public void init()
```

```
{ setLayout(new BorderLayout());
  button = new Button("Click Me");
  add("Center", button);
  button.addActionListener(this);
}

public void actionPerformed(
   (ActionEvent e)
{ Toolkit.getDefaultToolkit().beep();
}
}
```

Oft nutzt man die neuen inneren Klassen für die Eventbehandlung:

```
public class Beeper1
    extends Applet
{ Button button;
  public void init()
  { setLayout(new BorderLayout());
    button = new Button("Click Me");
    add("Center", button);
    button.addActionListener(
        new BeepAction());
  }

  class BeepAction
    implements ActionListener
  { public void actionPerformed(
       (ActionEvent e)
    { Toolkit.getDefaultToolkit().beep();
    }
  }
}
```

Wenn nur ein Handler-Objekt benötigt wird: anonyme Klassen :

```
public class Beeper2
    extends Applet
{ Button button;
```

```
public void init()
{
    setLayout(new BorderLayout());
    button = new Button("Click Me");
    add("Center", button);
    button.addActionListener(
        new ActionListener()
        {
            public void actionPerformed(
               (ActionEvent e)
            {
                Toolkit.getDefaultToolkit().
                    beep();
            }
        }
    );
}
```

12.1.1 Zum Vergleich: das alte Ereignisbehandlungsmodell

Nicht objektorientiert. Anwender macht Fallunterscheidung über Event-Typen und Event-Ziele:

```
public class Beeper3
    extends Applet
{
    // JDK 1.0 Event Handling
    Button button;
    public void init()
    {
        setLayout(new BorderLayout());
        button = new Button("Click Me");
        add("Center", button);
    }
    public boolean action(Event e,
                           Object arg)
    {
        if (e.target.equals(button))
            Toolkit.getDefaultToolkit().beep();
        else return super.action(e, arg);
        return true;
    }
}
```

12.1.2 Adapterklassen

Wenn ein Listener-Interface mehrere Methoden fordert (z.B. `MouseListener`), man aber nur an einer Art von Event interessiert ist, ist es lästig, alle geforderten Methoden (mit leerem Rumpf) zu implementieren.

Adapterklassen lösen das Problem, indem sie für alle Methoden des Interfaces leere Default-Implementierungen bereitstellen. Man erbt von einer Adapterklasse und überschreibt die benötigte Eventhandling-Methode:

```
MyClass extends MouseAdapter
{
    ....
    public void mouseClicked(
        MouseEvent e)
    {
        ...
    }
}
...
button.addMouseListener(new MyClass());
```

12.1.3 Listener und Behandlungsmethoden

Für alle Eventarten mit mehr als einer Behandlungsmethode existiert eine Adapterklasse entsprechenden Namens.

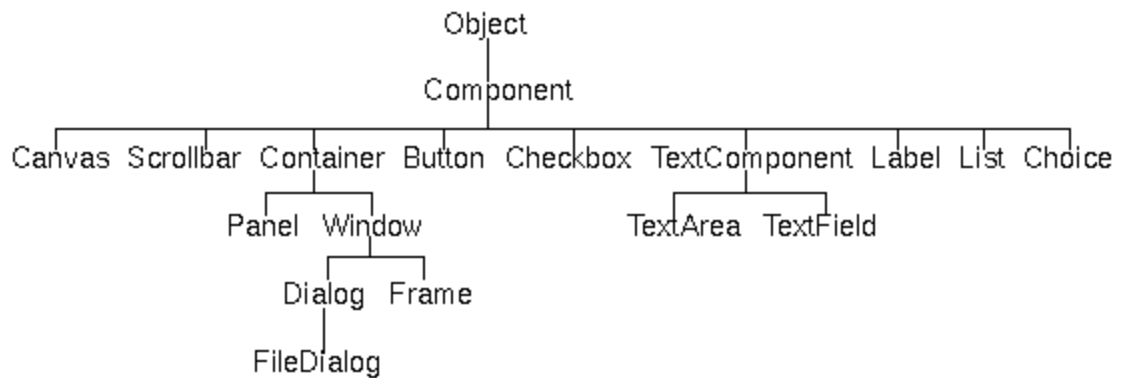
Beispiel: Listener: `KeyListener` Adapter: `KeyAdapter`

Listener Interface	Methoden
ActionListener	actionPerformed
AdjustmentListener	adjustmentValueChanged
ComponentListener	componentHidden
	componentMoved
	componentResized
	componentShown
ContainerListener	componentAdded
	componentRemoved
FocusListener	focusGained
	focusLost
ItemListener	itemStateChanged
KeyListener	keyPressed
	keyReleased
	keyTyped
MouseListener	mouseClicked
	mouseEntered
	mouseExited
	mousePressed
	mouseReleased

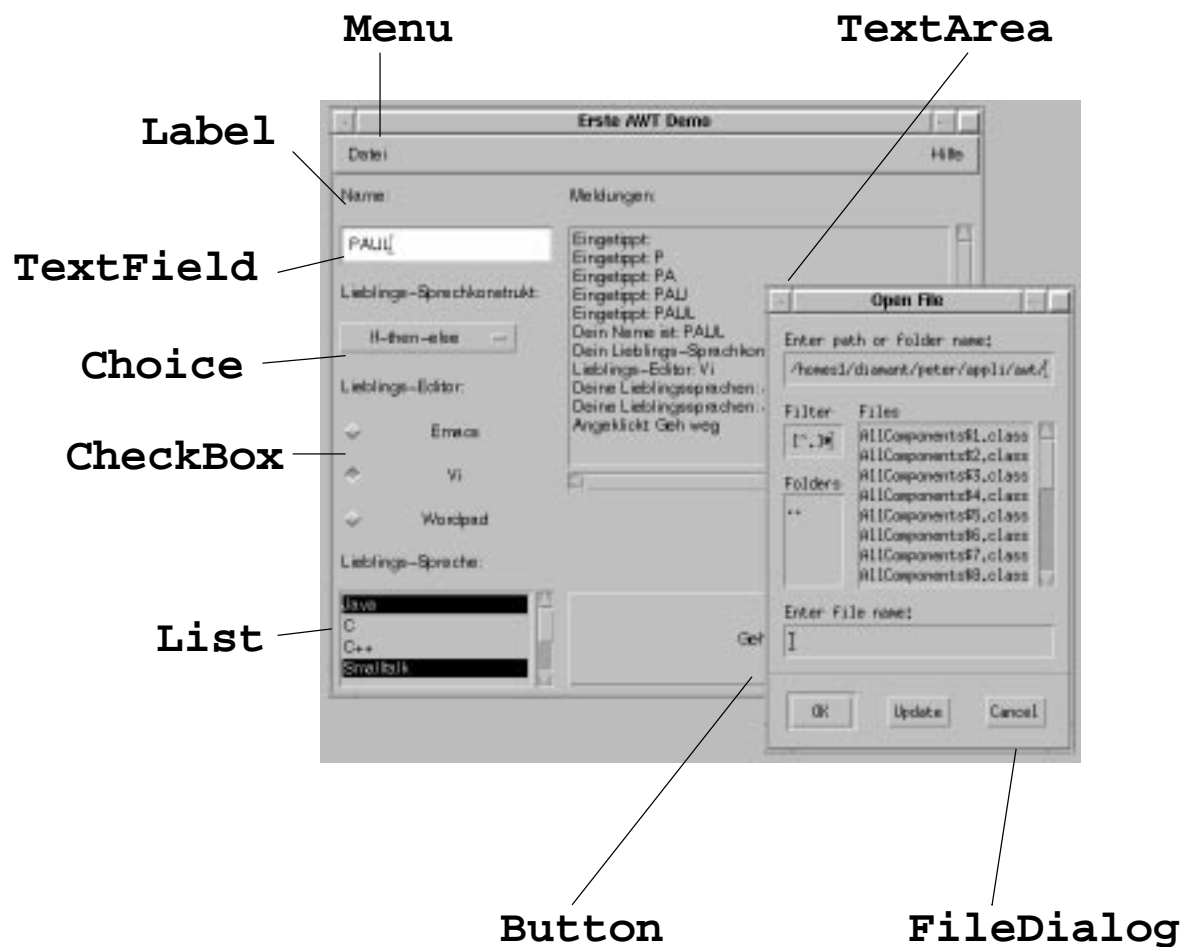
Listener Interface	Methoden
MouseMotionListener	mouseDragged
	mouseMoved
TextListener	textValueChanged
WindowListener	windowActivated
	windowClosed
	windowClosing
	windowDeactivated
	windowDeiconified
	windowIconified
	windowOpened

12.2 Die Komponenten im Java AWT

Unterklassen der Klasse Component



Diese Oberfläche enthält die meisten vordefinierten Komponenten des AWT:



12.2.1 Button

Die Klasse `Button` stellt Schaltflächen mit Aufschrift zur Verfügung. Klicken erzeugt einen `ActionEvent`.

```
Button cancel = new Button("Geh weg");
this.add(cancel);
// Event-Handling dafür
ActionListener buttonlistener =
    new ActionListener()
    { public void actionPerformed(
        ActionEvent e)
        { // Behandle Ereignis
          ...
        }
    };
cancel.addActionListener(buttonlistener);
```

12.2.2 Canvas

Freie rechteckige Fläche zum Zeichnen. Wird auch zur Erzeugung eigener Grafik-Komponenten verwendet. Zur Darstellung des `Canvas`:: `paint`-Methode überschreiben.

12.2.3 Checkbox

Grafische Schaltkomponente, die entweder ein- oder ausgeschaltet sein kann (`true` oder `false`). Klicken schaltet zwischen den beiden Zuständen hin und her.

```
add(new Checkbox("one", true));
add(new Checkbox("two"));
add(new Checkbox("three"));
```

Radioknöpfe (nur einer ist angeschaltet), können mit der Klasse `CheckboxGroup` realisiert werden:

```
CheckboxGroup ckg = new CheckboxGroup();  
add(new Checkbox("one", ckg, true));  
add(new Checkbox("two", ckg));  
add(new Checkbox("three", ckg));
```

12.2.4 Choice

Die Klasse `Choice` realisiert ein Pop-Up-Menü zur Auswahl. Die aktuell gültige Auswahl wird als Titel des Menüs angezeigt.

```
Choice ColorChooser = new Choice();  
ColorChooser.add("while");  
ColorChooser.add("If-then-else");  
ColorChooser.add("switch");
```

12.2.5 Label

Komponente, mit der man Read-Only-Text in einem Container plazieren kann.

```
add(new Label("Hi There!"));  
add(new Label("Another Label"));
```

12.2.6 List

Listen von Texteinträgen, aus denen ein oder mehrere ausgewählt werden können.

```
List lst = new List(3, false);
```

```
lst.add("Apfel");  
lst.add("Birne");  
lst.add("Kirsche");  
lst.add("Banane");  
container.add(lst);
```

Der zweite Parameter des Konstruktors steuert, ob mehrere Einträge auswählbar sind.

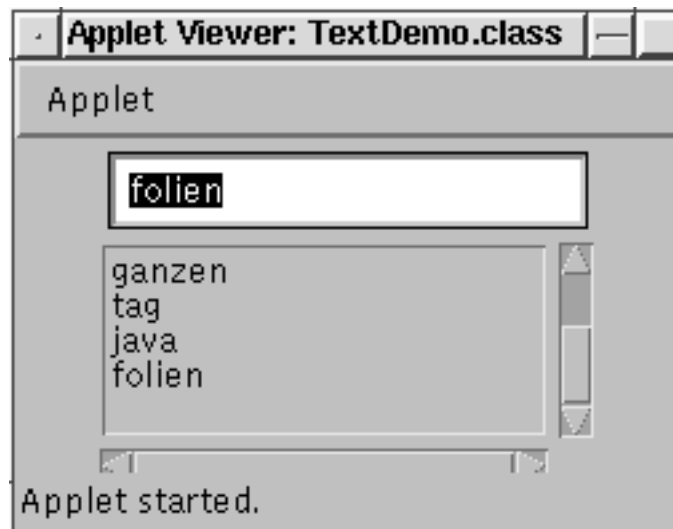
12.2.7 TextAreas und TextFields

Eine TextArea-Komponente ist ein mehrzeiliger Bereich in dem Text angezeigt und/oder eingegeben werden kann.

Eine TextField-Komponente ist ein einzeliger Bereich zur Texteingabe.

Beispiel:

```
// Import-Klauseln gelöscht  
public class TextDemo  
extends Applet  
implements ActionListener  
{ TextField zeile;  
  TextArea feld;  
  public void init()  
  { zeile = new TextField(20);  
    feld = new TextArea(5, 20);  
    feld.setEditable(false);  
    add(zeile);  
    add(feld);  
    zeile.addActionListener(this);  
  }  
  public void actionPerformed(  
   (ActionEvent evt)  
  { feld.append(zeile.getText()  
    + "\n");  
    zeile.selectAll();  
  }  
}
```



12.3 Die Container im Java AWT

Container sind Grafikkomponenten, die andere Komponenten aufnehmen können.

Zugefügte Komponenten (Methode `add()`), werden in einer Liste gespeichert und von einem **Layout Manager** angeordnet.

12.3.1 Panel

Einfachste Container-Klasse. Ein Bereich, in dem die Applikation andere Komponenten anordnen kann.

12.3.2 ScrollPane

Container-Klasse, die horizontales und vertikales Scrollen für eine enthaltene Komponente zur Verfügung stellt.

12.3.3 Window

Top-Level-Fenster ohne Rahmen oder Menu. Kann zum Beispiel zur Implementierung von Pop-Up-Menüs verwendet werden.

Solange es offen ist, blockiert es die Eingabe in andere Fenster.

12.3.4 Dialog und FileDialog

Unterklasse von Window zur Gestaltung von Benutzer-Dialogen bzw. Dateiauswahl.

Dialoge hängen von anderen Fenstern ab, sie werden z.B. zerstört wenn das Grundfenster zerstört wird.

12.3.5 Frame

Top-Level-Fenster mit Titel und Rahmen. Kann Menüs haben. Jede Applikation braucht mindestens einen Frame.

Aus dem Beispiel von Seite 167

```
public class AllComponents
    extends Frame
    implements ActionListener
{
    public AllComponents(String title)
    {
        super(title);
        // Reaktion auf Fensterschließen
        this.addWindowListener(
```

```
        new WindowAdapter()
        { public void windowClosing(
              WindowEvent e)
          { System.exit(0);
          }
        }
    );
    public static void main(String[] args)
    { Frame f = new AllComponents(
          "Erste AWT Demo");
      f.pack();
      f.show();
    }
}
```

12.4 Menüs

Menüs werden aus den Komponenten

- MenuBar: eine Menü-Leiste
- Menu: ein Menü
- MenuItem: ein Menü-Eintrag

konstruiert:

```
// Menü-Leiste
mb = new MenuBar();
setMenuBar(mb);

// Ein Menü
m1 = new Menu("Datei");
mb.add(m1);
```

```
// Menüeinträge  
schl = new MenuItem("Schließen");  
m1.add(schl);  
  
// usw.
```

Menü-Aktionen lösen Action-Events aus:

```
// Eintrag mit Shortcut  
MenuItem open =  
    new MenuItem("Öffne",  
                 new MenuShortcut(  
                     KeyEvent.VK_O));  
open.setActionCommand("open");  
open.addActionListener(this);  
  
....  
public void actionPerformed(  
   (ActionEvent e)  
{ String command = e.getActionCommand();  
  if (command.equals("open"))  
  { ...
```

Nur Container, die die MenuContainer-Schnittstelle implementieren können Menüs haben (z.B. ein Frame).

12.5 Layout Manager

Jeder Container hat einen Standard Layout Manager, der dafür zuständig ist, die enthaltenen Komponenten anzuordnen. Dieser voreingestellte Layout Manager kann durch einen anderen ersetzt werden.

Das AWT bietet:

- Einfache Layout Manager (FlowLayout und GridLayout)
- Spezielle Layout Manager (BorderLayout und CardLayout)
- Einen allgemeinen Layout Manager (GridBagLayout)
- Die Möglichkeit, eigene Layout Manager zu schreiben.

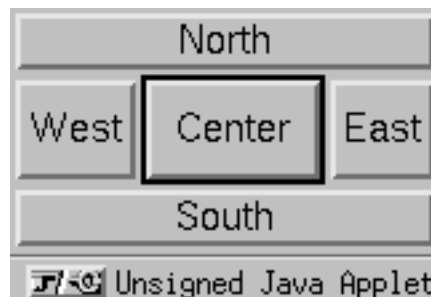
Beispiel

Zuordnung eines Layout-Managers zu einem Container:

```
meinContainer.setLayout(new CardLayout());
```

12.5.1 BorderLayout

Anordnung der Komponenten an den vier Rändern und in der Mitte.



```
import java.awt.*;
import java.applet.Applet;
public class ButtonDir extends Applet {
    public void init() {
        setLayout(new BorderLayout());
        add("North", new Button("North"));
        add("South", new Button("South"));
        add("East", new Button("East"));
    }
}
```



```

        add("West",    new Button("West"));
        add("Center",  new Button("Center"));
    }
}

```

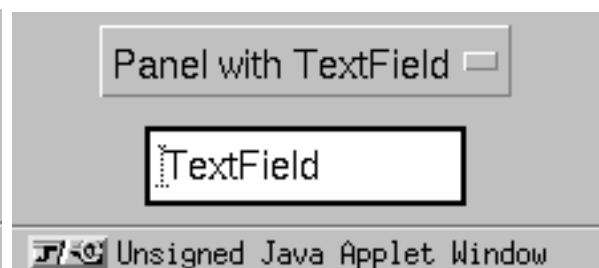
12.5.2 CardLayout

Überlagerung der Komponenten.

entweder



oder



```

cards = new Panel();
cards.setLayout(new CardLayout());
...// Erzeuge Panel p1 mit Knöpfen
...// Erzeuge Panel p1 mit Textfeld
cards.add("Panel with Buttons", p1);
cards.add("Panel with TextField", p2);

```

Die Strings dienen zur Identifikation der Karten. Das Umschalten zwischen sich verdeckenden Karten muß man ausprogrammieren, z.B.:

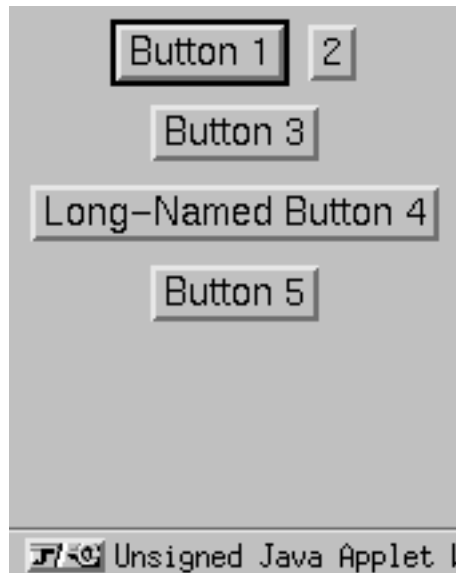
```

(CardLayout)cards.getLayout().
    show(cards, "Panel with TextField");

```

12.5.3 FlowLayout

Komponenten der Reihe nach angeordnet.



```
setLayout(new FlowLayout());  
setFont(new Font("Helvetica",  
                 Font.PLAIN, 14));  
add(new Button("Button 1"));  
add(new Button("2"));  
add(new Button("Button 3"));  
add(new Button("Long-Named Button 4"));  
add(new Button("Button 5"));
```

Durch Konstruktorargument kann angegeben werden, ob die Komponenten innerhalb einer Zeile zentriert (Default), rechts- oder linksbündig angeordnet werden.

12.5.4 GridLayout

Komponenten in einem Matrix-Raster.



```
// Beliebigviele Reihen, 2 Spalten
setLayout(new GridLayout(0,2));
add(new Button("Button 1"));
add(new Button("2"));
add(new Button("Button 3"));
add(new Button("Long-Named Button 4"));
add(new Button("Button 5"));
```

12.5.5 GridBagLayout

Komponenten in einem flexiblen Matrix-Raster. Mächtigster aber auch kompliziertester Layout Manager:

- Zeilen sind verschieden hoch
- Komponenten können über mehrere Zeilen gehen
- Komponenten können über mehrere Spalten gehen
- Resizing-Verhalten über Gewichte individuell einstellbar



Das Layout wird durch `GridBagConstraints` spezifiziert:

```
GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints c = new GridBagConstraints();
setLayout(gridbag);
```

```
//...Nach Setzen der Attribute von c:
add(irgendeineKomponente, c);
```

Die wichtigsten Attribute von `GridBagConstraints` sind:

- `gridx` und `gridy`: Zeilen- und Spaltenposition der Komponente.
- `gridwidth` und `gridheight`: Wieviele Spalten und Zeilen überspannt die Komponente.
- `fill`: Füllverhalten, wenn der Platz größer als die Komponente ist.
- `weightx` und `weighty`: Gewichte zwischen 0.0 und 1.0. Relativer Platzanspruch der Komponente in ihrer Zeile bzw. Spalte.

Die Knöpfe 8 und 9 der Oberfläche (Seite 178) wurden z.B. wie folgt erzeugt:

```
GridBagLayout griba = new GridBagLayout();
GridBagConstraints c =
    new GridBagConstraints();
setLayout(griba);
```

```
c.fill = GridBagConstraints.BOTH;

c.gridwidth = 1;
c.gridheight = 2;
c.weighty = 1.0;
add(new Button("Button8"), c);

c.weighty = 0.0;
c.gridwidth =
GridBagConstraints.REMAINDER;
c.gridheight = 1;

add(new Button("Button9"), c);
```

Inhaltsverzeichnis

1	Objektorientierte Programmierung	2
1.1	Abstraktionen in Programmiersprachen	3
1.2	Schnittstelle und Implementierung	5
1.3	Wiederverwendung	6
1.3.1	Wiederverwendung durch Komposition	7
1.3.2	Wiederverwendung durch Vererbung	7
1.4	Polymorphie	9
1.5	Lebensraum und Lebensdauer von Objekten	12
1.5.1	Wo leben Objekte?	12
1.5.2	Objektzugriff über Referenzen	13
1.5.3	Container und Iteratoren	16
1.6	Generische Typen	16
2	Ein Streifzug durch Java	18
2.1	Applets und Applikationen	18
2.2	Hello World	20
2.3	Variablen und Ablaufstrukturen	20
2.4	Kommentare	21
2.5	Benannte Konstanten	22
2.6	Klassen und Objekte	22
2.6.1	Erzeugen von Objekten	23
2.6.2	Klassenvariablen = Statische Attribute	24
2.6.3	Methoden und Parameter	24
2.6.4	Methodenaufruf	25
2.6.5	Selbstreferenz <code>this</code>	25
2.6.6	Klassenmethoden = Statische Methoden	26
2.7	Arrays	26
2.8	String Objekte	27

2.9	Erweitern von Klassen: Vererbung	28
2.10	Schnittstellen (interfaces)	29
2.11	Ausnahmen (Exceptions)	30
2.12	Pakete (Packages)	33
3	Klassen und Objekte	35
3.1	Konstruktoren	36
3.1.1	Die Selbstreferenz <code>this</code> in Konstruktoren	37
3.2	Klassenvariablen	39
3.3	Klassenmethoden	39
3.4	Initialisierer	41
3.4.1	Initialisierung von Klassenvariablen	42
3.4.2	Initialisierung von Instanzvariablen	43
3.5	Objekterzeugung	44
3.5.1	Speicherallokation	46
3.5.2	Default-Initialisierung	46
3.5.3	Konstruktion von Objekten	47
3.6	Objektzerstörung	48
3.6.1	Finalisierung von Objekten	49
3.7	Innere Klassen	49
3.7.1	Geschachtelte “top-level” Klassen und Interfaces	50
3.7.2	Elementklassen	52
3.7.3	Lokale Klassen	62
3.7.4	Anonyme Klassen	63
3.8	Sichtbarkeitsspezifikationen	64
4	Vererbung	66
4.1	Vererbung: Wann und wie?	68
4.2	Oberklassen-Konstruktion	69
4.3	Was wird vererbt?	70

4.4	Verdecken von Attributen	71
4.5	Verdecken durch Klassenmethoden	72
4.6	Überschreiben durch Instanzmethoden	72
4.7	Zusätzliche Regeln für Verdeckung und Überschreiben von Methoden	73
4.8	Dynamische Bindung	74
4.9	Überladen von Methoden	75
4.10	Die <code>final</code> -Kennzeichnung	81
4.10.1	Finale Daten	81
4.10.2	Finale Methoden	83
4.10.3	Finale Klassen	84
4.11	Abstrakte Methoden und abstrakte Klassen	84
5	Interfaces	86
5.1	Beispiel einer Interface-Anwendung	87
5.2	Einfacherbung gegen Mehrfacherbung	89
5.3	Interface-Deklarationen	90
5.4	Konstanten in Interfaces	91
5.5	Mehrdeutigkeiten bei Konstanten	92
5.6	Methoden in Interfaces	92
5.7	Beispiele für Interfaces der Java-Bibliothek	93
6	Arrays	95
6.1	Erzeugung von Arrays	95
6.2	Mehrdimensionale Arrays	96
6.3	Zugriff auf Arrays	97
7	Grundsymbole, Operatoren und Ausdrücke	98
7.1	Unicode-Buchstaben	98
7.2	Kommentare	99
7.3	Bezeichner	100

7.4	Reservierte Wörter	100
7.5	Grundtypen	101
7.6	Literale	101
7.7	Deklaration von Variablen	102
7.8	Initialisierung	103
7.9	Namensanalyse	105
7.10	Konventionen für die Verwendung von Bezeichnern	106
7.11	Operatorpräzedenz und -assoziativität	109
7.12	Auswertungsreihenfolge	110
7.13	Typanpassung	111
7.13.1	Implizite Typanpassung	111
7.13.2	Explizite Typanpassung	112
7.14	Operatoren	115
7.14.1	Arithmetik	115
7.14.2	String-Konkatenation	115
7.14.3	Inkrement/Dekrement	116
7.14.4	Vergleich und logische Operatoren	116
7.14.5	Bit-Operationen	117
7.14.6	Zuweisungen	118
8	Anweisungen	119
8.1	Verzweigung	119
8.2	Fallunterscheidung	120
8.3	while- und do-Schleifen	120
8.4	for-Schleife	121
8.5	Abbruch-Anweisungen: break, continue, return	121
8.5.1	Die break-Anweisung	122
8.5.2	Die continue-Anweisung	123
8.5.3	Die return-Anweisung	123

<i>INHALTSVERZEICHNIS</i>	186
9 Ausnahmen	124
9.1 Definieren von Ausnahmetypen	124
9.2 Auslösen von Ausnahmen	125
9.3 Abfangen von Ausnahmen	126
10 Parallelausführung	130
10.1 Starten von Threads	131
10.2 Zustand eines Threads	135
10.3 Synchronisation von Threads über Daten	135
10.3.1 Synchronisierte Methoden	136
10.3.2 Synchronisierte Anweisungen	137
10.4 Synchronisation von Threads mit <code>wait</code> und <code>notify</code>	138
11 Ein-/Ausgabe	142
11.1 Byte-Ströme	143
11.1.1 Binäre Eingabe-Ströme	143
11.1.2 Binäre Ausgabe-Ströme	148
11.2 Zeichen-Ströme	150
11.2.1 Zeichen-Eingabe-Ströme	150
11.2.2 Zeichen-Ausgabe-Ströme	152
11.3 Random Access Dateien	153
11.4 Datei- und Verzeichnis-Namen	154
11.5 Zerlegung von Strömen in Einzelsymbole	156
11.6 Ausgabe von Objekten (Serialisierung)	156
12 Grafische Benutzungsschnittstellen	161
12.1 Das Ereignis-Modell des AWT (ab JDK1.1)	162
12.1.1 Zum Vergleich: das alte Ereignisbehandlungsmodell	164
12.1.2 Adapterklassen	165
12.1.3 Listener und Behandlungsmethoden	165

12.2 Die Komponenten im Java AWT	166
12.2.1 Button	168
12.2.2 Canvas	168
12.2.3 Checkbox	168
12.2.4 Choice	169
12.2.5 Label	169
12.2.6 List	169
12.2.7 TextAreas und TextFields	170
12.3 Die Container im Java AWT	171
12.3.1 Panel	171
12.3.2 ScrollPane	172
12.3.3 Window	172
12.3.4 Dialog und FileDialog	172
12.3.5 Frame	172
12.4 Menüs	173
12.5 Layout Manager	174
12.5.1 BorderLayout	175
12.5.2 CardLayout	176
12.5.3 FlowLayout	177
12.5.4 GridLayout	177
12.5.5 GridBagLayout	178