

# **Einführung in Java**

Martin Steffen

Sommersemester 1998

Praktikum (Java)

# Lektion I

## Einführung

**Inhalt:** Einführung · erste Schritte · Programmierumgebung  
· einfachste Verwendung der Bibliothek

---

## Was ist Java?

- einfach
  - “in C it is easy to shoot yourself in the foot, in C++ it is easy to blow your whole leg away” (Bjarne Stroustrup)
- objektorientiert (klassenbasiert)
- interpretiert
- robust und sicher
- plattformunabhängig und portabel
- effizient (?)
- multithreaded
- implizite Speicherverwaltung (garbage collection)
  - “explicit memory management has proved to be a fruitful source of bugs, crashes, memory leaks and poor performance” (Sun Microsystems, 1995, *The Java Language Environment, A White Paper*)

---

## Historisches

Folgendes nach P. Naughton

- **Jan. 91:** „*Stealth project*“: Erste Vorbesprechungen (Joy, Gosling, Naughton . . . )
- **Juni 91:** Arbeit am **Oak**-Interpreter beginnt (Gosling), später umbenannt in **Java**
- **August 91:** Präsentation erster Ideen vor den Sun-Bossen
- **März 93:** Hauptstoßrichtung der Oak-Entwicklung wird *interaktives Fernsehen*, (Sun will einen Auftrag von Time-Warner an Land ziehen)
- **April 93:** NSCA Mosaic verfügbar
- **Juni 93:** Der Deal mit Time-Warner kippt, SGI bekommt den Zuschlag
- **Sommer 93 – Frühjahr 94** Suche nach Anwendungen (Consumer-Electronics, Interaktives Fernsehen, OS für Set-Top-Boxen, CDs Online, Multimedia, OS (Liveoak) . . . )
- **Juli 94:** Entscheidung: “Anwendung” von *Liveoak* ist das **Internet**
- **September 94:** Prototypimplementierung des Browsers *Web-Runner* beginnt (später HotJava)
- **Herbst 94:** Bootstrapping: Javacompiler in Java

- 
- **Mai 1995:** Sun stellt Java und HotJava auf der *SunWorld '95* vor, Netscape springt auf und will Java lizenzieren
  - **später im Jahr:** weitere Firmen, darunter Microsoft, lizenzieren Java
  - **Dezember 1995:** *Javascript* von Sun/Netscape
  - ...
  - **Oktober 1997:** Beginn des ersten Java-“Krieges” (Netscape vs. Microsoft)

---

## Java: die Speerspitze der Programmiersprachen?

- Statisches Typkonzept
- Typsicherheit (?)
- virtuelle/abstrakte Maschinen
- objektorientiert
- Interfaces
- parametrische/generische Module, parametrische Polymorphie
  - “**Q:** Are there parameterized types (templates)?”
  - “**A:** Not in Java 1.0 or 1.1. However this is being seriously considered for future versions.”,
  - (Frequently asked question aus `comp.lang.java`)
- Typinferenz
- type casts
- automatische Speicherverwaltung (garbage collection)

---

# Programmierungsumgebung

- **Compiler** (javac). Sourcecode  $\mapsto$  Bytecode
- **Interpreter** (java): interpretiert Bytecode
- **Appletviewer** (appletviewer): Spezialform des Interpreters, interpretiert *Applets*, die in HTML-Seiten eingebunden sind.

## Beispiel 1 (Hello World)

---

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

---

- HelloWorld.java enthält die Definition der **Klasse** HelloWorld.
- javac HelloWorld.java liefert HelloWorld.class
- java HelloWorld führt HelloWorld.class aus.
- appletviewer mypage.html interpretiert alle in die Seite eingebundenen Applets

# Klassen und Objekte

Später werden wir genauer auf objektorientierte Programmstrukturierung eingehen, für's erste soll folgende Charakterisierung genügen:

- **Klasse**
  - Programmcode ist in Klassen organisiert.
  - reserviertes Wort `class`
  - `public`-Klasse bestimmt den Dateinamen (mit Extension `java`) (d.h., bei  $> 1$  Klassendefinition in einer Datei ist genau eine davon `public`, diese bestimmt den Dateinamen.)
  - **Application**: stand-alone Java-Programm, mit Methode `main` (`public static void`).
  - Klassen: **Definition von Objekten**, d.h., Vereinbarung von (Objekt-lokalen) Variablen und **Methoden**, aber keine Speicherreservierung.
- **Objekt**
  - Einheit von **Daten** und **Methoden**, die auf den Daten operieren.
  - **Instanz** einer Klassen
  - alle Instanzen einer Klasse: gleiche Methoden, aber eigener Speicherbereich und eigenen Werten der lokalen Variablen
  - `new` instantiiert eine Klasse in ein Objekt, d.h., gibt die Referenz auf das neue Objekt zurück



- 
- Objekte existieren zur Laufzeit, Klassen zur Compilezeit
  - Zugriff nur über **Methoden** (Kapselung).

## Java API

- *Application Programming Interface*: Javas Klassenbibliothek
- Organisiert in verschiedene Packages
- **Package**: Gruppe verwandter Klassen, hierarchisch aufgebaut (Bsp: package java enthält alle API-Pakete).
- Referenzierung: z.B. java.applet.

java.lang	Zentrale Klassen. Paket wird <b>automatisch</b> in jedes Javaprogramm importiert.
java.applet	Programmierung von <i>Applets</i>
java.awt	Graphikunterstützung, GUIs
java.beans	wiederverwendbare SW-Komponenten
java.io	Input/Output
java.math	mathematische Funktionen
java.net	Netzprogrammierung
java.rmi	<i>Remote method invocation</i>
java.security	(Netzwerk-) Sicherheit
java.sql	Arbeiten mit Datenbanken
java.text	Textformatierung
java.util	Sonstige nützliche Klassen (Datentypen)

Tabelle 1: Java API

## import Anweisung

- Abkürzung anstelle *vollqualifizierter* Schreibweise zur Referenzierung von Klassen
- Gebrauch
  - `import package.class`
  - `import package.*`
- das Paket `java.lang` ist grundsätzlich automatisch importiert. z.B.: die Klasse `System` kann ohne `import java.lang.*` verwendet werden:

$$\underbrace{\text{System}}_{\text{Klasse}} . \underbrace{\text{out}}_{\text{Konst.}} . \underbrace{\text{println}}_{\text{Methode}} (\underbrace{\text{"Text"}}_{\text{akt. Param.}});$$

- Beispiel: siehe nächste Folie

**Beispiel 2 (Importieren)** Durch Importieren der Klassen aus dem Paket `java.io` hat man direkten Zugriff auf beispielsweise `BufferedReader`

```
import java.io.*;                                // I/O-Klassen
class Test_Class {

    public static void main (String [] args)
        throws IOException {
        BufferedReader stdin = new BufferedReader
            (new InputStreamReader (System.in));
```

```
.....  
};
```

Anstelle alle Klassen im I/O-Paket auf einmal zu importieren, hätte man auch die zwei benötigten herauspicken können:

```
import java.io.BufferedReader;  
import java.io.InputStreamReader;  
...
```

---

## Applications und Applets

- **Applet**  $\neq$  stand-alone Programm = Applications
- keine main-Methode, Interpreter sucht stattdessen (u. a.) nach **paint**-Methode
- Abgeleitet (durch **Vererbung**) aus der Klasse Applet des Pakets java.applet (Vererbung kommt später genauer)
- Applets werden (i. d. R.) in HTML-Seiten eingebettet und durch Netbrowser interpretiert.

**Beispiel 3 (Applet)** Ein sehr einfaches Beispiel für ein Applet, genauer gesagt, eine *Klasse*, deren *Instanzen* Applets sind. Es ist eine *Unterklasse* von Applet.

---

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

public class FirstApplet extends Applet {
    public void paint (Graphics page) {
        page.drawString ("Hello World", 50, 50);
    }                                   // Methode paint
}                                     // Klasse FirstApplet
```

---

## Lektion II

### Daten- & Kontrollstrukturen

**Inhalt:** Struktur der Sprache · primitive Datentypen  
· Schleifen · bedingte Anweisungen, etc.

**Literatur:** Einen hervorragenden Schnelleinstieg bietet *Java in a Nutshell* [Fla97b]. Der folgende Abschnitt baut zu einem Teil auf diesem Buch auf (Kapitel 2). Wer es gemächlicher und lehrbuchartiger wünscht, ist mit *Java Software Solutions* [LL97] gut bedient

## Lexikalische Struktur

- Anzahl **reservierter Worte**, d.h. von der Sprache vorgelegte *Identifizier*. Schlüsselworte für primitive Datentypen, Kontrollstrukturen, "Modifizierer" . . . . Manche sind (noch) ohne Bedeutung
- **Kommentare:**

<code>// Rest der Zeile ist Kommentar</code>	wie in C++
<code>/* &lt;Kommentar&gt; */</code>	wie in C/C++
<code>/** Dokumentarkommentar */</code>	für javadoc

- **Unicode-Zeichen**<sup>1</sup> für Identifizier, Strings
- kein **Präprozessor** (also kein `#define <KONST>`, `#include <File>`, `#ifdef`)
- **String** für **Zeichenketten** ist kein primitiver Datentyp, aber man kann Stringlitterale wie gewohnt ("Hello World") verwenden

---

<sup>1</sup>ASCII und ISO-Latin-1 ("mit Umlauten") kompatibel

## Primitive Datentypen

- **Primitive Datentypen**: alles außer Klassen und Arrays
- Größe ist implementierungs*unabhängig*
- Uninitialisierte Variablen haben einen default-Wert für jeden dieser Typen<sup>2</sup>
- Operatoren auf den Datentypen, mit Überladen (Overloading)
- Java ist stark typisiert, also keine C-“Tricks” wie

```
if (i) ... else ..... ;  
while (i--) .....
```

wenn i ein Integer ist.

- Fließkomma-Literale: defaultmäßig double, aber 1234.5f oder 1234.5F erzwingt float

---

<sup>2</sup>aber sich darauf zu verlassen, ist kein guter Stil



Werte	Schlüsselwort	Bitlänge
numerisch ganze Zahlen, alle <i>mit</i> Vorzeichen	byte	8
	short	16
	int	32
	long	64
Fließkommazahl	float	32
	double	64
boolesch true o. false	boolean	1
Zeichen UNI-Code	char	16

- Deklaration/Definition beispielsweise

```
int    i    = 5;
boolean b = true;
char   c = 'f';
float  x = 1234.4f;    // !!
```

# Operatoren

- die Operatoren sind:
  - typisiert, und zum Teil überladen
  - (partiell) geordnet nach Präzedenz (Bindungsstärke)
  - mit unterschiedlicher Assoziation (links/rechts)
  - Meist Infix-Notation<sup>3</sup>
- ansonsten C-ähnlich (siehe ausgeteiltes Blatt), zusätzlich:
  - >>>: Rechtsschift mit 0
  - | und & auf boolean: *logisches* „oder“ bzw. „und“<sup>4</sup>
  - instanceof
- es fehlen Operationen für Pointerarithmetik \* (Dereferenzierung), & \\

---

<sup>3</sup>Der Operator ? beipiesweise für bedingte Ausdrücke (<cond>?<expr1> : <expr2>) ist dreistellig und *Mixfix*, Inkrement mit ++ ist einstellig und *Postfix*.

<sup>4</sup>Ansonsten *bitweise*

---

## Referenzdatentypen

- **Referenzdatentypen**: Objekte und Arrays
- Manipulation “by reference”, also über die Adresse/Referenz.
  1. *Variablen* eines Referenztyps speichern die Referenz, nicht den Wert. Wertzuweisung kopiert Referenz
  2. Übergabe der Referenz an *Methoden*
  3. == vergleicht, ob auf das selbe Object/Array verwiesen wird.
- **null**: Leere Referenz
- Garbage Collection
- Aber: keine **Dereferenzierung**, keine **Pointer**, kein **Cast** von Referenztypen zu primitiven Typen oder umgekehrt.

---

## Wrapper Klassen

- zu jedem primitiven Datentyp gibt es eine **Wrapper**-Klasse
- “das gleiche” wie die elementare Werte eines primitiven Datentyps, nur als **Objekte** einer **Klasse** (dem **wrapper**)
- Namenskonvention: ähnlich dem Namen des primitiven Datentyps, aber Großbuchstaben (Byte, Short, ..., Character)
- Beispiel: Umwandlung (kein cast):

```
int i = 15;  
Integer intobject = new Integer(i);
```

- Nutzen: An manchen Stellen wird ein **Objekt** erwartet, kein elementarer Wert<sup>5</sup>

---

<sup>5</sup>Es gibt noch einen zweiten Nutzen von Klassen wie Integer. Das kommt aber erst später.

## Arrays

- nicht-primitiver Datentyp: Objekte mit “Spezialsyntax”
- Array der Länge  $n$ : indiziert von 0 bis  $n - 1$
- Syntax:<sup>6</sup>

```
float[] height = new float [25];
height[0]      = 20.9f;           // nicht 20.9
System.out.println(height.length);
```

- Bound-check zur Laufzeit
- Instanzieren eines Arrays von Objekten: Speicherreservierung, keine Instantiierung der Objekte
- Initialisierung mit Initialisierungsliste:

```
String[] zahlen = {"eins", "zwei", "drei"};
```

- mehrdimensionale Arrays: Arrays von Arrays von...

```
int fourD [] [] [] [] = new int[10] [] [] [] ;
                        // new int[] [10] [] [] ; verboten
int twoD [] [] = {{1}, {1,2}, {1,2,3}};
```

---

<sup>6</sup> Alternativ ist auch `float height [] = ..` erlaubt, ist aber nicht konsistent mit der Vorstellung, daß die Syntax immer `<datatype> <ident> = new <datatype>` lautet.

---

# Strings

- kein primitiver Datentyp (sondern Klasse `Java.lang.String`), aber
  - es gibt **Stringlitterale** wie gewohnt
  - **Operator** (keine Methode) + zur Konkatenation von Strings
- implizite Konversion in einen String in Fällen<sup>7</sup> wie:  

```
System.out.println("Die Antwort lautet " + 42);
```

---

<sup>7</sup>Genauer gesagt: es funktioniert auch mit Objekten, die eine Methode `toString` zur Verfügung stellen.

## Einfacher Kontrollfluß

Block	Gruppierung mit { und }. ; als Trenner
bedingte Anweisungen	
if	<code>if (&lt;cond&gt;) &lt;stmnt1&gt; else &lt;stmnt2&gt;</code>
switch	<pre> switch (expr) {   case value1:     stmnt-list;   case value2:     stmnt-list;   .... }; </pre>
Schleifen	
while	<code>while (&lt;cond&gt;) &lt;stmnt&gt;;</code>
for	<code>for (&lt;init&gt;; &lt;cond&gt;; &lt;incr&gt;) &lt;stmnt&gt;;</code>
do	<code>do &lt;stmnt&gt; while (&lt;cond&gt;;</code>

- kein `goto`
- `switch`: *konstante* Ausdrücke, und zwar ganze Zahl oder char; optionaler `default`-Zweig am Ende. Für *exklusive* Auswahl: Zweige (bis auf default) mit `break` abschließen!
- `break`: Verlassen des Schleifenrumpfes<sup>8</sup>/des Switchzweiges
- `continue`: Überspringen des weiteren Schleifenrumpfes

<sup>8</sup>`break` in Schleifen wird manchmal als schlechter Stil angesehen.

- 
- Es gibt auch **gelabelte** break/continue — für die die goto's vermissen.



# Lektion III

## Objektorientierte Strukturen (1)

**Inhalt:** Objekte und Klassen · Vererbung · Klassenhierarchie  
· Überschreiben

**Literatur:** Die Lektion baut teilweise auf Kapitel 3 aus [Fla97b] auf. Siehe auch Kapitel 4 und 8 aus [LL97]. Eine schöne allgemeine Übersicht und Diskussion über objektorientierte Sprachmerkmale bietet das Buch *Object-Oriented Programming* [Bud97].

---

# Klassen

- **Klasse:** Definition von Objekten, d.h., die Vereinbarung von Daten und Methoden
- aller Programmcode ist in Klassen organisiert
- **Methode:**

```
<return_type> <method_name> ( <parm_list> ){  
    <stmt_list>  
};
```

- **Methoden-lokale Variablen:** nur innerhalb dieser Methode gültig.
- **return:** Rückgabe eines (typkonformen) Wertes (return <expr>;), **Kontrollfluß** kehrt an den Aufrufer zurück. Ein bloßes return gibt "nichts" zurück (Typ void).<sup>9</sup>

---

<sup>9</sup>Das gleiche gilt für eine Methode ohne return.

**Beispiel 4 (Circle)** Der *Zustand* eines Kreises als einer Instanz von `Circle` — die Koordinaten seines Zentrums und sein Radius — wird in den entsprechenden **Feldern** oder **Instanzvariablen** mit passendem Typ gespeichert.

---

```
public class Circle1 {  
  
    //          Felder  
    public double x, y;    // Daten = Koordinaten  
    public double r;      // in die Felder  
  
    //          Methoden  
  
    public double circumference() {  
        return 2 * 3.1415926 * r;  
    };                    //  
};                        // '{' und '}' zur Gruppierung
```

---

## Objekte

- **Objekt**: Einheit von Daten ("fields") und **Methoden**, die auf den Daten arbeiten.
- **Instanz** einer Klasse (Klasse als "Datentyp" des Objektes).
- Neue Objekte: Instantiierung mittels **new**
- **Methoden/Feld-Selektion** ("Methodenaufruf): mittels **"."**.
- implizites Argument **this**: Instanz an der die Methode aufgerufen wird
- implizites **this** auch in Methodendefinition: in Beispiel 4: `return 2 * 3.1415926 * r` entspricht `return 2 * 3.1415926 * this.r` (später mehr)

**Beispiel 5 (Circle)** Zugriff auf den Zustand eines Kreisobjektes = Instanz von `Circle1` über seine Felder oder Methoden

---

```
Circle1 c = new Circle1();    // Instantiierung

c.x = 4.0;                    // Zugriff auf Felder
c.y = 2.3;                    // und Methoden mit
c.r = 1.5;                    // '.'

double a = c.circumference();
```

---

---

## Objekte und Klassen: Konstruktoren

- **Konstruktor**: “Spezialmethode” zur **Initialisierung** eines Objektes mit dem selben Bezeichner wie die Klasse
- `new`  $\Rightarrow$  Aufruf der **Konstruktor**-Methode.
- Jede Klasse besitzt (mindestens einen) Konstruktor selben Namens, falls keiner angegeben wird  $\Rightarrow$  **Default**-Konstruktor ohne Parameter (siehe vorheriges Beispiel 5)
- mehrere Konstruktoren mit unterschiedliche Parameterlisten möglich: Beispiel für “**method overloading**”.<sup>10</sup>
- kein (expliziter) Rückgabotyp

---

<sup>10</sup> Geht auch bei normalen Methoden, d.h. nicht-Konstruktormethoden.

**Beispiel 6 (Circle)** Hier das gleiche Beispiel nochmal, diesmal mit (nicht-default) Konstruktor. *Instantiierung* nun mittels `Circle2 c = new Circle2(3.2, 4.5, 1.9)`.

---

```
public class Circle2 {
    private double x, y;    // private: kommt sp"ater
    private double r;      // genauer
    Circle2 (double _x, double _y, double _r) {
        x = _x;
        y = _y;
        r = _r;
    };                      // Konstruktor Circle2
    //-----
    public double circumference() {
        return 2 * Math.PI * r;    // Konstante der Klasse Math
    };
};
```

---

## Instanz- und Klassenvariablen

Variablendeklarationen in einer Klasse (Felder) kann man in Instanz- und Klassenvariablen unterteilen

1. bisher und für gewöhnlich: **Instanzvariable**: eine Kopie pro Instanz
2. **Klassenvariable** oder auch *statische Variable*:
  - eine Kopie der Variable pro Klasse
  - Schlüsselwort **static**
  - Zugriff: ebenfalls mittels '.', allerdings über den *Klassennamen*: `<classname>.<var_name>`<sup>11</sup>
  - sie entsprechen **globale** Variablen mit dem Klassennamen als *Diskriminator* (z.B. `Math.PI`)<sup>12</sup>
  - **lokale** Variablen (innerhalb einer Methode) dürfen nicht statisch sein

---

<sup>11</sup> Instanzen der betrachteten Klasse dürfen den Klassennamen weglassen.

<sup>12</sup> `static final` anstelle von `#define` ....

**Beispiel 7 (Anzahl der Instanzen)** Sei ein Kurs durch die Menge seiner Teilnehmer modelliert. Die Klassenvariable `freiePlaetze` drückt einen *globalen* “Zustand” des Kurses aus.<sup>13</sup>

---

```
public class Teilnehmer {
    public static int freiePlaetze = 10;
    public String name;
    public int matrikelnummer;

    Teilnehmer(String n, int m) {
        name = n;
        matrikelnummer = m;
        freiePlaetze--;           // einer weniger frei
    };                           // Konstruktor
};
```

---

---

<sup>13</sup>Wie man eine Überbuchung des Kurses verhindert, kommt später.



---

## Klassenmethoden, statische Methoden

Analog der Situation bei Feldern unterscheidet man **Instanz-** und **Klassenmethoden**.

1. bisher (fast) nur **Instanzmethoden**
2. **Klassenmethoden** oder auch *statische Methoden*
  - Schlüsselwort **static** (“Modifier”)
  - Zugriff: über den **Klassenbezeichner** (analog Klassenvariablen)
  - Wichtig: **kein** impliziter **this**-Parameter!
  - Beispiel: die Klasse `java.lang.Math` enthält nur statische Methoden<sup>14</sup>

---

<sup>14</sup>Grund: `Math` hat viel mit Zahlen zu schaffen, und das sind Werte, keine Objekte.

## Initialisierung von Klassenvariablen

- Neben der *Standardsyntax* (Initialisierung bei der *Variablen-deklaration*, z.B. `static int x = 5; ...`) gibt es für komplexere Initialisierungen: **static initializers**
- Code, automatisch beim **Laden** der Klasse ausgeführt<sup>15</sup>
- **namenlose** Methode, **kein** Rückgabetyt (implizit `void`)
- mehrere derartige Initialisierungsfragmente möglich
- **Syntax:**

```
<modifier> class Class_name {  
    ...  
    static {  
        <Initialisierungs-code>  
    };  
    .....  
};
```

- häufige Verwendung: für Klassen mit **native**-Methoden
- neben *statischen* Initialisierern gibt es auch **Instanzinitialisierer**: gleiche Syntax, aber ohne `static`-Schlüsselwort. (Später mehr)

---

<sup>15</sup> natürlich nicht bei Instantiierung eines zugehörigen Objektes.

---

## Lebensende von Objekten: Müllabfuhr und Finalisierung

- implizite, automatische Speicherverwaltung: **Garbage Collection** (in Ruhephasen und bei Bedarf)
- **“Finalization”**: spezielle Methode (`finalize()`) von Objekten, aufgerufen bevor der Speicher für das Objekt freigegeben wird.
- Verwendung: Freigabe von **Systememressourcen**, die nicht automatisch — wie der Speicher — verwaltet werden, z. B.: *Schließen von Dateien, Abbau von Netzverbindung* etc.

## Unterklassen und Vererbung

- Definition neuer Klassen unter Verwendung/Erweiterung bereits bestehender Klassen  $\Rightarrow$  Unterklasse, Klassenhierarchie

- Schlüsselwort **extends**:

```
<modifier> class Class_Name extends Class_Name' {  
    ....  
};
```

Class\_Name ist die Unterklasse, Class\_Name' die Oberklasse (subclass, superclass)

- Schlüsselwort **final** nicht-erweiterbare Klasse
- Unterklasse:
  - **Erbt**: Felder und Methoden der Oberklasse (Ausnahme solche, die **private** sind)
  - **Überschreiben** (overriding), d.h. Reimplementieren einzelner Methoden möglich.
  - Verwendung der "alten" Methoden der Oberklasse: Referenzierung mittels **super**
  - Instanz der Unterklasse ist **auch** eine Instanz der Oberklasse und kann als solche verwendet werden  $\Rightarrow$  **Polymorphie**
  - Übernahme der (expliziten) **Konstruktoren**: zwingend mit **super**(<arg-list>), **super** folgt dann **unmittelbar** dem Konstruktor

---

## Beispiel 8 (Konto)

---

```
public class Konto {
    int    kontostand;
    String inhaber;
    Konto (String name, int i) {
        kontostand = i;
        inhaber = name;
    };
    Konto (String name) { // konstruktor-overloading
        kontostand = 0;
        inhaber    = name;
    };
    /* Konto (){
        kontostand = 0;
        inhaber    = "";
    };
    */
    //-----
    public void einzahlen (int betrag) {
        kontostand += betrag;
    };
    public void auszahlen (int betrag) {
        kontostand -= betrag;
    }

    public int wieviel () {
        return kontostand;
    };
};
```

---

**Beispiel 9 (Sparkonto)** Ein *Sparkonto* sei eine besondere Art von Konto, die auch noch *Verzinsung* bietet.

---

```
public class Sparkonto extends Konto { // Unterklasse
    double z_faktor;
    double ueberziehungsz_faktor;
    Sparkonto (double z_satz, double z_satz2, String name, int i) {
        super(name,i);
        z_faktor = 1+ z_satz /100;
        ueberziehungsz_faktor = 1+ z_satz2/100;
    };
    Sparkonto (double z_satz, String name) {
        super(name); // this.inhaber = name;
                    // w"are falsch
        z_faktor = 1 + z_satz/100;
    };
    /* Sparkonto(double rate) {... ginge nicht, denn
       super hat kein Konto()-Konstruktor */
    // -----
    public void verzinsen() {
        kontostand = (int) (z_faktor * kontostand);
    };

    public void auszahlen (int betrag) { // "Uberschreiben
        if (kontostand >= betrag)
            kontostand -= betrag;
    };
};
```

---

## Klassenhierarchie

- die Klassen von Java bzw. der Klassenbibliothek formen eine Hierarchie
  - mit Wurzel `java.lang.Object`
  - `final`-Klassen haben keine Unterklassen
  - Kette der Konstruktoren (*constructor chaining*, bei der Instanziierung; explizit oder implizit. Ausnahme: man umgeht es mittels `this`).
- bis auf `Object`: jede Klasse hat eine Oberklasse
  - explizit mittels `extends`
  - falls `extends` fehlt: Unterklasse von `Object`
- eine Klasse kann beliebig viele Interfaces implementieren. Beachte: dies ist *keine Mehrfachvererbung*

## Überschreiben von Methoden

- **Überschreiben** (*overriding*): eine Kerneigenschaft oo-er Sprachen
- *dynamischer Methodenaufruf, dynamic dispatch, späte<sup>16</sup> Bindung, message passing*:  
Methodenaufruf läßt sich als Prozeduraufruf mit später Bindung verstehen
- **static**-, **private**- und **final**-Methoden können nicht überschrieben werden
- Referenzierung der überschriebenen Methode und der überschreibenden Klasse mit **super**: `super.<methode>`
- „überschriebene“ Felder: (*shadowed*), am besten vermeiden

---

<sup>16</sup>d.h. zur Laufzeit



---

## Beispiel 10 (Überschreiben)

---

```
class A {
    int i = 1;
    int f() { return i;}
};

public class B extends A {
    int i;
    int f() {
        System.out.println("i = " + i);
        i = super.i + 1;
        System.out.println("i = " + i);
        System.out.println("super.f() = " + super.f());
        return super.f() + i;
    };
};
```

---

## Kapselung

- Wichtiges OO-Merkmal: Zugriff auf Daten (möglichst) nur über Methoden
  - Datenabstraktion, Strukturierungs- und Scoping-Konzept . . .

erlaubter Zugriff	Sichtbarkeit			
	public	protected	default	private
selbe Klasse	Ja	Ja	Ja	Ja
selbes Paket	Ja	Ja	Ja	Nein
Unterklassen (anderes Paket)	Ja	Ja	Nein	Nein
beliebige Klasse	Ja	Nein	Nein	Nein

Tabelle 2: Sichtbarkeitsmodifikatoren

- Merke: Geerbte Variablen und Methoden **behalten** ihre Sichtbarkeitsstufe
- **Konstruktoren** sind **public** (und werden nicht vererbt)
- **protected**: werden an Unterklassen **vererbt**

---

## Kapselung (2)

Ein paar **Daumenregeln** für die verschiedenen Sichtbarkeitsstufen

- **public**: für Methoden und Konstanten, die der Benutzer sehen soll. Sehr sparsam bei Feldern<sup>17</sup>
- **protected**: Unwichtig für die *Benutzung* der Objekte, aber wichtig für jemand, der ein Paket um Unterklassen außerhalb des Paketes **erweitert**
- **default** (auch Paket-Sichtbarkeit): Methoden, die zur Kooperation innerhalb eines Paketes notwendig sind
- **private**: lokale verborgene Definitionen innerhalb einer Klasse

---

<sup>17</sup> Am besten, man vermeidet es ganz indem man *Zugriffsmethoden* schreibt.

---

## Abstrakte Klassen

- **abstakte Klasse:** nicht instantiierbar (**abstract**)
- **abstrakte Methode:** ohne Rumpf. nur Signatur
- **Zusammenhang:**
  - “konkrete” Klasse mit abstrakten Methoden: **verboten**
  - abstrakte Klasse mit “konkreten” Methoden: **erlaubt**
  - konkrete **Unterklasse** einer abstrakten Klasse muß die abstrakten Methoden (typkonform) **implementieren**<sup>18</sup>
- **Nutzen:**

---

<sup>18</sup>Eine Sonderform des *Überschreibens*.

---

## Selbstreferenz: `this`

# Lektion IV

## Ausnahmebehandlung

**Inhalt:** Verwendung von Ausnahmen · Definition eigener Ausnahmebehandlung

**Literatur:** Den entsprechenden Abschnitt aus [Fla97b] oder Abschnitt 14.1 aus [LL97]. Daneben unter anderem die Klassen `java.lang.Throwable`, `java.lang.Error`, `java.lang.Exception.*`

## Einführung

- nicht alle *unerwünschten* Situationen können statisch überprüft und abgefangen werden ⇒ **Exceptions**
- *disziplinierte* Art, den **Kontrollfluß** abhängig vom (Fehler- ) Verhalten zu steuern
- Zur **Steuerung**:
  - **throw**: **Signalisierung** einer **Ausnahmesituation** ⇒ Verlassen des gewöhnlichen Kontrollflusses
  - **catch**: **Definition der Reaktion** auf eine **Ausnahme**
- **Nutzen**:
  - saubere **Trennung** von Normal- und Ausnahmeverhalten, d.h. Ausnahmebehandlung getrennt vom Rest anpassbar (vorteilhaft auch bei der Verwendung fremder Klassen/APIs)
  - Debugging einfacher
  - Typsicherheit

---

## Beispiel 11 (Division durch Null)

---

```
class Div {
    public static int  division (int z, int n) {
        return z/n;
    };
};

public class Main {
    public static void main (String[] args) {
        int nenner = 0;
        int zaehler = 0;
        System.out.println(Div.division(zaehler, nenner)); // Zeile 11
    };
};
```

---

Ergibt als Ausgabe folgende Aufrufstruktur auf dem Stack:

```
java.lang.ArithmeticException: / by zero
    at Div.division(Main.java:3)
    at Main.main(Main.java:11)
```

Process Main exited abnormally with code 1

Die Ausnahme ist eine Instanz von `ArithmeticException`, einer Klasse aus `java.lang`.



## Ausnahmen

- **Ausnahmen** sind **Objekte**, genauer Instanzen von `java.lang.Exception` (bzw. deren Unterklassen)
- sie implementieren das **Interface** `Throwable`<sup>19</sup>
- Propagieren einer geworfenen Ausnahme durch die *lexikalische Blockstruktur* bis
  - nach *ganz oben* (`main`) oder
  - die Ausnahme **abgefangen** und **behandelt** wird (mit `catch`)
- **Syntax**: `try ... catch .... [finally]`.

```
try {
    <stmnt_list>
} catch <except_class1> <var1> {
    <stmnt-list_1>}{
    catch <except_class2> <var2> {
        <stmnt-list_>2}{
    ....
}
finally
{
    <stmnt_list_n>
}
```

---

<sup>19</sup>Kommt später genauer, was das heißt

- `try`: führt ein Statement „unter Vorbehalt“ aus
- `catch`: fängt eine geworfene Ausnahme bei Bedarf ab und reagiert darauf, der erste passende `catch`-Zweig behandelt eine eventuelle Ausnahme.
- optional `finally`: wird grundsätzlich ausgeführt (falls vorhanden)

**Beispiel 12 (Division durch Null (II))** Der folgende Stück Code zeigt, wie man den Divisionsfehler abfangen — hier eher in nicht-empfehlenswerter Weise ignorieren — kann:<sup>20</sup>

---

```
class Div {
    public static int  division (int z, int n) {
        return z/n;
    };
};
public class Main2 {
    public static void main (String[] args) {
        int nenner    = 0;
        int zaehler   = 0;
        int ergebnis = 0;                // vorbelegen
        try {
            ergebnis = Div.division(zaehler, nenner);
        }
        catch (ArithmeticException e) {
            System.out.println("Mir egal, ich rechne weiter");
        };
        System.out.println(ergebnis);
    };
};
```

---

Ein klein wenig *informativer* wäre bereits die Ausgabe

```
System.out.println(e + ", trotzdem rechne ich weiter");
```

---

<sup>20</sup>Wäre ergebnis nicht bei der Deklaration bereits ein Wert zugewiesen worden, hätte der Compiler erkennen können, daß ich schummele.

## Erzeugen und Definieren von Ausnahmen

- Neben Abfangen und behandeln kann man Ausnahmen auch
  - **Deklarieren**: `throws`. Kann die Ausführung einer Methode zu einer Ausnahme führen, die **nicht abgefangen** wird: Deklaration im „Rückgabety“

```
public void open_file() throws IOException {
    .....                // der Rumpf der Methode
                          // kann scheitern
};
public void mymethod(..) throws MyExcpt1, MyExcpt2 {
    .....
};
```
  - **Generieren** = *werfen* mittels `throw`
  - **Definieren**: da, wie erwähnt, Ausnahmen **Objekte** sind werden sie als *Klassen*, genauer als **Unterklassen** von `Exception` vereinbart
  - **Parameterübergabe** zur Fehlerdiagnose (oft ein `String`) mittels der **Konstruktoren**

---

## Beispiel 13 (Ausnahmen)

---

```
class MyException extends Exception {
    int wert;
    MyException(int i) {                // Konstruktor
        wert = i;
    };
    public int get_wert() {
        return wert;
    };
};
public class Main {
    public static void main (String[] args) {
        int x;
        try
            {Main.unsafe_method(0);}
        catch (MyException e) {
            System.out.println("MyException mit wert = " +
e.get_wert());};
    };
    public static void unsafe_method(int i) throws MyException {
        if (i == 0)    {throw (new MyException (i));}
        else System.out.println("OK");
    };
};
```

---

# Lektion V

## Objektorientierte Strukturen (2)

**Inhalt:** Interfaces · Pakete · innere Klassen

**Literatur:** Während im ersten OO-Abschnitt Merkmale von Java behandelt wurden, die in ähnlicher Form in vielen anderen objektorientierten Sprachen auch zu finden sind, geht es hier um mehr Java-spezifische Dinge. Nachlesen kann man es in Kapitel 5 aus [Fla97b]. Über Interfaces und Pakete findet sich auch in [LL97] etwas.

---

## Dies & Das

Hier einige Dinge, die ich nicht erzählt habe und die wir (bis auf die Casts) vermutlich nicht brauchen, beziehungsweise die später kommen.

- Vorwärtz-Referenzen erlaubt
- Schlüsselworte: `native`, `transient` `volatile`, `synchronized`
- `final`-Modifikator:
  - Klassen: keine Unterklassen
  - Methoden: nicht überschreibbar
  - Variablen (kein Schreibzugriff). In `Java 1.1` auch anwendbar auf Methodenparameter und lokale Variablen

## Dies & Das (2): Typumwandlungen

- **Typumwandlungen**: ein Programmfragment kann mehrere "Typen" besitzen, in Objekt Instanz mehrerer Klassen sein (instanceof) ⇒ Polymorphie
- nur zwischen Referenztypen oder zwischen nicht-Referenztypen.
- Bei elementaren Datentypen
  - **Narrowing**: Informationsverlust, z.B: von float nach int)
  - **Widening**: umgedreht
- drei Arten
  - in Wertzuweisungen und Methodenaufrufen: implizit, nur Widening erlaubt (Subsumption)
  - Arithmetische Promotion (z.B. in 3.0 / 3)
  - Explizite Umwandlung (**type casts**):
    - \* Schönheitsfehler im Typsystem: Laufzeittypüberprüfungen
    - \* Syntax: (type) value



---

## Beispiel 14 (Casts)

```
.....  
Konto      k = new Konto("Tick", 500);  
Konto      k2 = new Konto("Trick", 1000);  
Sparkonto  sk = new Sparkonto(10, "Track");  
  
k = sk;                // Impl. Umwandlung des sk  
  
((Sparkonto)k).verzinsen(); // Cast  
((Sparkonto)k2).verzinsen(); // Cast
```

# Interfaces

- **Interface**: enthält **abstrakte Methoden**<sup>21</sup> und **Konstanten** (`static final`)
- Default: Methoden des Interfaces sind `public`
- nicht **instanzierbar**
- $\cong$  **Signatur** der Instanzen einer Klasse
- Klasse implementiert ein Interface (**implements**)
  - muß **typkonforme Implementierung** aller **Methoden** liefern<sup>22</sup>
  - **Konstanten** werden “geerbt”
- mehrere Interfaces erlaubt, aber:  $\neq$  **Mehrfachvererbung**
- Kombination mit **Unterklassenbildung** möglich
- **Syntax**  
`<class1> implements <interf1>, <interf2> extends <class2>`
- Verwendung als Typ eines formalen Parameters: `public void mymethod (MyInterface arg) {.....}`
- **Subinterfaces** (mittels **extends**). Wichtig wiederum: ein Interface kann **mehrere** Interfaces erweitern

---

<sup>21</sup> das Schlüsselwort `abstract` muß nicht angegeben werden, ist aber implizit vorhanden.

<sup>22</sup> Bei mehreren Interfaces: alle Methoden. Falls die Interfaces eine Hierarchie bilden, auch noch deren darüberliegenden Interfaces.

## Pakete

- dreistufige Hierarchie: **Pakete**  $\Rightarrow$  Klassen  $\Rightarrow$  Mitglieder<sup>23</sup>
- Konvention: Paketnamen oft mit "." separiert und Paketnamen = Verzeichnispfad
- Vorschlag zur weltweiten Namensvergabe von Klassen:<sup>24</sup>

`de.uni – kiel.informatik.users.ms.classes.class, meth`  
Paket, Verzeichnis users/ms/classes *Klasse Mitglied*

- Schlüsselwort **package**: zu Beginn einer Quelldatei, gibt die Zugehörigkeit der Klasse zum Paket an
- **import**: relativ zum CLASSPATH (import package.\* oder import package.class)

---

<sup>23</sup>Members = bisher Methoden und Felder.

<sup>24</sup>Interfaces werden, was Zugriff und Namensresolution betrifft, genauso behandelt wie Klassen, deswegen werden sie hier nicht extra genannt.

- Zugriffsrechte: (Siehe auch Tabelle 2 auf Folie 41)
  - Paket: über Zugriffsrechte (File/Netz)
  - Klassen im Paket: zugreifbar für alle anderen Klassen im Paket, `public`-Klassen auch außerhalb des Paktees
  - Klassenmitglieder: innerhalb des `Paketes`, es sei denn sie sind `private`
  - Innerhalb einer Klasse: freier Zugriff der Mitglieder untereinander.
  - Klassenmitglieder einer Klassen *A* zugreifbar in *B* eines anderen `Paketes`, falls
    - \* *A* `public` und das Mitglied `public`, oder
    - \* *B* Unterklasse von *A* und das Mitglied `protected`

---

## Innere Klassen

- **innere Klassen**: wichtige Spracherweiterung in JDK 1.1
- oft verwendet in den 1.1-API-Klassen, vor allem im **Event-Modell**
- bisher: Klassen definierbar in genau **einem Kontext**: an oberster Stufe in einem **Paket**<sup>25</sup>
- **Vier neue Arten von Klassen = vier neue Kontexte**
  1. **geschachtelte Top-level Klassen**:
  2. **nicht-top-level Klassen**
    - (a) Klassen in Klassen: **member Klassen**
    - (b) Klassen in Methoden: **lokale Klassen**
    - (c) Klassen ohne Namen: **anonyme Klassen**
- keine Änderung des **Laufzeitsystems** (*Java Virtual Machine*), d.h. **syntaktischer Zucker**, Quellcodetransformation

---

<sup>25</sup>Wenn kein Paket angegeben, dann an oberster Stufe in einem anonymen default Paket

---

## Geschachtelte Top-level Klassen/Interface

- “statische” Klasse innerhalb einer anderen, als Mitglied
- `static`-Modifikator
- geschachtelte Interfaces sind implizit *statisch*
- Semantik: genau wie andere Klassen, nur der Name der Klasse enthält die umgebende Klasse.
- $\Rightarrow$  feinere hierarchische Aufteilung des Namensraumes für Klassen, weniger Namenskonflikte
- geht für Klassen und Interfaces
- benutzung von `import` wie bei Paketmitgliedern: `import Klasse_aussen.Klasse_innen`

## Member Klassen

- **Member-Klasse**: nicht-statisches Mitglied einer äußeren Klasse (mit anderem Namen)
- darf keine **statischen** Mitglieder enthalten
- **Interfaces** nicht möglich (immer statisch)
- Instanzen sind mit einer Instanz der umgebenden Klasse **assoziiert** ⇒ Zugriff auf deren **Instanzmitglieder**, d.h. des Objektes
- Verwendung: **Hilfsklassen** der äußeren Klasse
- Problem:
  - worauf bezieht sich **this**? Antwort: auf die Instanz der “inneren” Klasse ⇒ neue **Syntax**:  
`Klasse_aussen.this.methode`
  - welches ist die *assoziierte* äußere Instanz? Antwort: Im Normalfall die aktuelle Instanz (**this**) der äußeren Klasse. Auch möglich:  
`aeussere_Instance.new innere_Klasse(...)`
- member-Klassen sind **schachtelbar**
- etwas Abartiges: Top-level Klasse kann eine Memberklasse **erweitern**<sup>26</sup>

---

<sup>26</sup>Um damit umzugehen, wird auch die `super`-Syntax entsprechend erweitert. Vor dem Gebrauch wird abgeraten

- Member-Klassen: Gültigkeitshierarchie, getrennt von der Unterklassenhierarchie
- Sichtbarkeiten: wie üblich (`public` allerdings selten)

**Beispiel 15 (Member)** Folgendes Beispiel zeigt das Absuchen einer Liste unter Verwendung `java.util.Enumeration`. Wollten wir den `Enumerator` nicht als `Member`, sondern als getrennte top-level Klasse definieren, müssten wir ihm die `LinkedList` als `Argument` des Konstruktors mitgeben. Hier kann direkt mit `head` direkt auf das Feld der umgebenden Klasse zugegriffen werden.



```
import java.util.*;
public class LinkedList4 {
    public interface Linkable removeHead() { // top-level nested
        public Linkable getNext();
        public void setNext(Linkable node)
    };

    Linkable head;
    public void addToHead(Linkable node) {...};
    public Linkable removeHead() {...};

    public Enumeration enumerate() {return new Enumerator()};
    private class Enumerator implements Enumeration { // Member!
        Linkable current;
        public Enumerator() {current = head;} // Konstruktor
        // -----
        // 2 Methoden des Enumeration-interfaces
        public boolean hasMoreElements() { return (current != null);}
        public Object nextElement (){
            if (current == null)
                throw NoSuchElementException("LinkedList");
            Object value = current;
            return value;
        }
    }
}
}
```

---

---

## Lokale Klassen

- innere Klasse innerhalb eines Blockes
- sichtbar nur innerhalb des definierenden Blocks, analog *lokalen Variablen*, beispielsweise eine Klasse *innerhalb einer Methode*
- können auf Mitglieder der *umschließenden* Klasse zugreifen, insbesondere auf `final` lokale Variable und Parameter innerhalb des Blockes<sup>27</sup>
- Verwendung of als sog. *Adapterklassen*, speziell im *Eventmodell* von Java 1.1 (*Event Listener* als Implementierung für *listener nterfaces*)

---

<sup>27</sup> technische Bemerkung: `final` deswegen, weil der Compiler *Kopien* für die lokale Klasse anfertigt.

---

## Beispiel 16 (Lokale Klasse)

---

```
import java.util.*;
public class LinkedList4 {
    public interface Linkable removeHead() { // top-level nested
        public Linkable getNext();
        public void setNext(Linkable node)};

    Linkable head;
    public void addToHead(Linkable node) {...};
    public Linkable removeHead() {...};

    public Enumeration enumerate() {
        class Enumerator implements Enumeration {
            Linkable current;
            public Enumerator() {this.current =
                LinkedList.this.head;}
            // -----
            // 2 Methoden des Enumeration-interfaces
            public boolean hasMoreElements() { return (current != null);}
            public Object nextElement (){
if (current == null)
    throw NoSuchElementException("LinkedList");
Object value = current;
current = current.getNext();
return value;
        }
    }
}
```



---

## Anonyme Klassen

- **anonyme Klasse** = namenslose lokale Klasse
- ein einziges Mal instanziiierbar
- Kombination von **Klassendefinition** und **Instantiierung** in einem **Ausdruck** (keine Anweisung)
- **Syntax** (innerhalb eines Ausdruckes), beispielsweise: die anonyme Klasse implementiert ein *Interface*

```
... new <interface> { <Klassendef.> } ..
```

**Beispiel 17 (Anonyme Klasse)** Im folgendes Beispiel implementiert eine anonyme Klasse das *Interface* `FilenameFilter` (mit Methode `accept`) aus dem Paket `java.io`.<sup>28</sup>

---

```
import java.io.*;

public class Lister{

    public static void main(String[] args) {
        File f = new File(args[0]);
        String[] list = f.list(new FilenameFilter() {
            public boolean accept(File f, String s) {
                return s.endsWith(".java");
            }
        }); // Ende der anon. Klasse

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

---

<sup>28</sup>Man kann ausprobieren, was der Compiler draus macht: er erzeugt eine neue Klasse `Lister$1.class`

## Klassenarten: Überblick

Klasse	Beschreibung		
top-level	Paketmitglied	K/I	Java-1.0, direkt interpretierbar durch die VM
	geschachtelt	K/I	<code>static</code> innerhalb einer anderen top-level Klasse
Innere Klasse	Member-Klasse	K	nicht-statisches Mitglied, besitzt umschließende <i>Instanz</i> , mit Zugriff auf deren Mitglieder, neue Syntax für <code>new</code> , <code>super</code> , <code>this</code> . Statische Mitglieder verboten
	Lokale Klasse	K	innerhalb eines <code>Blocks</code> , Zugriff auf Mitglieder der umschließenden Klasse + <code>final</code> lokale Variable, neue <code>new</code> -Syntax
	Anonyme Klasse	K	<code>namenlose</code> lokale Klasse, <i>eine</i> Instanz, kein Konstruktor

Tabelle 3: Arten von Klassen

# Lektion VI

## Input/Output

**Inhalt:** Zu jeder Sprache gehören Fähigkeiten zur Datenein- und ausgabe. Neben den umfangreichen Interaktionsmöglichkeiten auf GUI/Event-basis, stellt die Klassenbibliothek auch Ein- und Ausgabe über *Ströme* zur Verfügung.

**Literatur:** Die Rohinformation steht natürlich in den Klassen des Paketes `java.io.*`. Relevant daneben Abschnitt 3.3 aus [LL97].

**Bemerkung:** Da wir nun in die Klassen der API einsteigen, wird es von nun an in der Vorlesung nicht mehr möglich sein, mehr oder minder das gesamte Material — alle Klassen und deren Methoden — vorzustellen oder auch nur zu erwähnen. Für die Lösung der Aufgaben wird es deswegen zum Teil notwendig



---

sein, die Klassenbibliothek selbstständig zu Rate zu ziehen. Die Klassenbibliothek ist online verfügbar. Drei gedruckte Exemplare sind am Lehrstuhl kurzfristig ausleihbar.

---

## Verschiedenes

- I/O nicht im Java-Kern, sondern mittels geeigneter Klassen.  
relevantes Paket: `java.io`
- einfachste Ausgabe auf stdout: `System.out.println(...)`<sup>29</sup>
- Quelle für Input oder Output: **Strom** (*stream*)
- drei **vordefinierte** Ströme:
  1. `System.in` (meist Keyboard)
  2. `System.out` (meist Screen)
  3. `System.err` (meist Screen)
- **Escape** mittels `\` (Liste `\b \t \n \r \" \' \\`)
- Input- oder Output-**Puffer**: temporärer Speicher
- **Leeren** des Ausgabepuffers: *flushing*, explizit durch `flush`, meist durch `println`, nicht aber durch `print`.

---

<sup>29</sup>`out` ist eine "Konstante" (`public static final`) aus `java.lang.System`.

## Übersicht über java.io.\*

- Aufgaben
  - Lesen/Schreiben von **Dateien**
  - Lesen und Informationsgewinnung von **Verzeichnissen**
  - Bereitstellung von vordefinierten **Strömen**
  - Unterstützung bei der **Definition** eigener Ströme
- **Strom**: Objekt, welches **sequentielles** Lesen und Schreiben erlaubt.
- die Klasse enthält in der Hauptsache **viele Stromklassen**
- **vier** Hauptvertreter, siehe Tabelle 4, die spezielleren Klassen sind jeweils **Unterklassen** davon.
- `System.in` : `InputStream`, und `System.out` und `System.err`: `PrintStream`

	Input	Output
Byte	<code>InputStream</code>	<code>OutputStream</code>
Character	<code>Reader</code>	<code>Writer</code>

Tabelle 4: Übersicht über Stromklassen

## Ein/Ausgabe: wichtige Klassen

- `InputStreamReader`: Konversion von byte nach character: Übersetzung nach Unicode
- `BufferedReader`: Liest einen Character-Puffer<sup>30</sup>
- `InputStream`: Oberklasse aller input-stream, byte-stream
- `FileInputStream`: Lesen von Bytes oder Arrays von Bytes von einem File (= Instanz von `File`).<sup>31</sup>

```
File from_file = new File(from_name); // from_name: String
File to_file   = new File(to_name);
...
```

```
FileInputStream from = null;
FileOutputStream to   = null;
try {
    // I/O kann immer schiefgehen
    from = new FileInputStream(from_file);
    to   = new FileOutputStream(to_file);
    ...
}
```

- `FileOutputStream`: Analog zum Schreiben
- Sonstiges: Verbinden von Strömen, Komprimieren, Filtern, etcetc

---

<sup>30</sup> Pufferung wg. Effizienz

<sup>31</sup> -1 als EOF-Rückgabe

**Beispiel 18 (Buffered reader)** Das Beispiel zeigt, wie man *Text vom stdin* lesen kann. Um von `System.in` lesen zu können, muß man die Eingabe “puffern”

---

```
import java.io.*;

class Echo {

    public static void main (String[] args) throws IOException {
        BufferedReader stdin =
            new BufferedReader (new InputStreamReader (System.in));

        String message;

        System.out.println("Bitte String eingeben: ");
        message = stdin.readLine();    // Meth. des BufferedReader

        System.out.println("Echo: \"" + message + "\"");
    };
};
```

---

---

## Nicht-Stromklassen

- `File` die wichtigste nicht-Stromklasse
  - Representiert den `Namen` eines Files/Verzeichnisses
  - stellt Fileoperationen darauf zur Verfügung (Kopieren, Umbenennen, Auflisten des Verzeichnisses ... )
- Wahlfreier ( $\neq$  sequentieller) Zugriff: `RandomAccessFile`
- `Exeptions`: `IOExceptions` und Unterklassen für speziellere I/O-Ausnahmen

# Lektion VII

## Applets und Graphik

**Inhalt:** Es geht um die Programmierung von Applets, das Zeichnen von Graphiken, Parameterübergabe. (Animation und Threads, die man für die Animation braucht, kommt später, falls noch Zeit ist, genaueres)

**Literatur:** Die Rohinformation zu diesem Abschnitt bieten die Klassen aus dem Paket `java.applet` und aus `java.awt.Graphics`. Dazu Kapitel 6 aus [Fla97b], Kapitel 4 aus [Fla97a] oder Kapitel 7 aus [LL97].

## Allgemeines

Zu ersten Anmerkungen zu *Applets*, siehe auch die einführende Lektion zu Beginn.

- **Applets:** Unterklassen von `java.Applet`
- Applets stehen unter der Kontrolle des *Appletviewers/Browsers*
  - ⇒ keine `main()`-Methode notwendig
  - ⇒ eine Reihe von Standardmethoden, die der Appletviewer aufrufen kann, wie z.B. `paint` und die überschrieben werden
  - ⇒ Applet muß auf diese Methodenaufrufe prompt antworten, d.h., für *Animationen* benötigt es `Threads`.
- Hierarchie:

```
Object => Component => Container => Panel => Applet
```

d.h., viele (auch) für Applets wichtige Methoden finden sich in vor allem `Component`

- Sonstiges:
  - Sicherheitseinschränkungen für Applets
  - digitale *Signaturen*



## Wichtige Methoden bei Applets

Name	Ausführung bei	typische Verwendung
init()	Laden des Applets	Initialisierung, Parameterübergabe. Funktion ähnlich einem Konstruktor
destroy()	Unload	Recourcenfreigabe
start()	Sichtbar-werden	Malen, Starten der Animation
stop()	Unsichtbar-werden	Unterbrechen der Animation
getAppletInfo()		Darstellbar in Dialogboxen
getParameterInfo()		Parameter des Applets
Aus Oberklassen (Object => Component => Container => Panel => Applet)		
repaint(_)		(Component), ruft update auf, welches den Bildschirm löscht und paint aufruft
paint(Grapics)		Malen (Container)
print(Graphics)		Drucken

Tabelle 5: Wichtige Methoden "an" Applets

---

## Applets-Lebenszyklus

### Beispiel 19

---

```
import java.awt.Graphics;

public class Lebenszyklus extends java.applet.Applet {
    StringBuffer buffer = new StringBuffer();

    public void init () {resize(500, 20);addItem("Initializing...");};
    public void start () {addItem("Starting ...");};
    public void stop(){addItem("Stop...");};
    public void destroy(){addItem("preparing for unloading");};

    public void addItem(String meldung) {
        System.out.println(meldung);
        buffer.append(meldung);
        repaint();
    };

    public void paint(Graphics g) {
        g.drawRect(0,0,size().width -1, size().height -1);
        g.drawString(buffer.toString(), 5, 15);
    };
};
```

## Häufig in Applets verwendete Methoden

Die Klasse *Applet* stellt auch Methoden zur Verfügung, die man häufig in Applets brauchen kann.

Image	<code>getImage(URL)</code>	Laden von Bildern
String	<code>getParameter(String)</code>	für Parameterübergabe aus HTML-Seite
URL	<code>getDocumentBase()</code>	Url der HTML-Seite
URL	<code>getCodeBase()</code>	Url der Klasse
	...	

Tabelle 6: In Applet definierte Methoden (Auswahl)

---

## Appletparameter

- Parameterübergabe aus HTML-Seiten
- Lesen der Parameter in der `init()`-Methode, mittels `getParameter(<param_string>)`
- HTML-Fragment:

```
<applet code=myclass.class width=200 height=200>  
<param name="bild" value="meinbild.gif">  
....  
</applet>
```

## Applets als Stand-alone Programme

- Applets müssen immer irgendwo eingebettet sein
- dies wird jetzt nicht vom *Appletviewer/Browser* gemacht:  
⇒ `Frame` selber erzeugen (auch Größe setzen)<sup>32</sup>
- Darstelle des Frames: `show`
- Aufrufen der `Applet-init`-Methode durch `main`

**Beispiel 20 (Standalone Applet)** Erweitert (und vereinfacht) aus [Fla97a]

---

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

public class StandaloneScribble extends Applet {
    public static void main (String[] args) {
        Frame f = new Frame();
        Applet a = new StandaloneScribble();
        f.add(a, "Center");
        a.init();
        f.setSize(400, 400);
        f.show();           // show l"a"st einen Frame erscheine
        f.setBackground(bgcolor);
        f.addWindowListener(new WindowAdapter() {
```

---

<sup>32</sup>die graphische Komponente `Frame` kommt später genauer, es ist eine Art top-level Fenster.

```
        public void windowClosing(WindowEvent e) {System.exit(0);}
    });
};          // end of main

public void init() {          // auch als applet verwendbar
    this.addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent e) {
            lastx = e.getX();          //
            lasty = e.getY();
        }
    });

    this.addMouseMotionListener(new MouseMotionAdapter() {
        public void mouseDragged(MouseEvent e) {
            Graphics g = getGraphics ();
            int x = e.getX();
            int y = e.getY();
            g.setColor(StandaloneScribble.this.drawcolor);
// wiederum: this alleine geht nicht!
            g.drawLine(lastx, lasty, x, y);
            lastx = x; lasty = y;
        }
    });

    Button b = new Button("Blau");
    b.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            drawcolor = Color.blue;
        }
    });

    this.add(b);
}
```

```
};  
  
protected int lastx, lasty;  
protected static Color bgcolor = Color.white;  
protected Color drawcolor = bgcolor;  
};
```

---

## Graphik-Objekte

- Instanzen von `java.awt.Graphic` sind “bemalbare” Oberflächen mit kartesischen Pixel-Koordinaten<sup>33</sup>
- i.d.R. nicht direkt instantiierbar<sup>34</sup>
- neue Graphik:
  - `Component.getGraphics()/Image.getGraphics(_)`
  - Kopieren: `Graphics.create(_)`
- Methoden zum Zeichnen verschiedener Dinge<sup>35</sup> (Strings, Linien, Ellipsen, Rechtecke . . . )
- zwei Modi: Normal oder XOR (zweimal XOR löscht wieder)

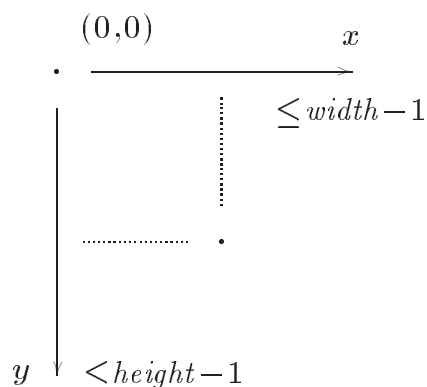


Abbildung 1: Koordinatensystem

<sup>33</sup>die Implementierung der Graphiken ist deviceabhängig, Instanzen von `Graphic` stellen also ein deviceunabhängige Schnittstelle zur Verfügung.

<sup>34</sup>Konstruktor ist `protected`

<sup>35</sup>Bitte selber in der Klasse nachschauen, was es so gibt.



---

## Graphische Elemente: Fonts, Farben & Cursor

- **Font** in der Klasse `java.awt.Font`
  - drei Bestimmungsstücke als Konstruktorparameter: `Name`, `Stil`, `Größe` (`String`, `int`, `int`)
  - `FontMetrics`: Bestimmung verschiedene `Maße` für Fonts
  - `setFont`: (in `Graphic`) setzt den aktuellen Font einer Graphik
- **Farben** in der Klasse `java.awt.Color`
  - `vordefinierte` Farben als “Konstanten” (`public static final`)
  - Konstruktoren mit `RGB-Parameter` (*rot-grün-blau*)
  - Umrechnung in `HSB-Farbmodell` möglich
- **Cursor** in der Klasse `java.awt.Cursor`
  - Verschiedene Standardcursors, insbesondere `DEFAULT_CURSOR`.
  - Übergebbar der Methode `setCursor(Cursor:c)`

## Lektion VIII

# Graphische Benutzerschnittstellen (GUIs)

**Inhalt:** Übersicht · “Components” · Layout

**Literatur:** Dieser Abschnitt diskutiert Auszüge aus dem Paket `java.awt`. Das Paket ist sehr groß, sodaß wir nur einen kleinen Teil ansprechen können. Bestimmte Teile (Graphiken, Fonts, Farben) wurden auch bereits in einem vorangegangenen Abschnitt besprochen)

Weitere Informationen finden sich in Kapitel 8 von [Fla97b], Kapitel 6 aus [Fla97a] oder Kapitel 10 aus [LL97]. Events sind eng mit der Programmierung von GUIs verbunden, kommen aber später noch genauer.

---

# Übersicht über das Abstract Windowing Toolkit

Wir können vier wichtige Gruppen von Klassen aus dem AWT ausmachen

- **Graphik:** Klasse `Graphics` zum Pixelmalen (siehe vorangegangenen Abschnitt)
- **Komponenten:** Klasse `Component` (und `MenuComponent`) und ihre Unterklassen: Interfaceelemente zur Interaktion
  - Knöpfe
  - Listen
  - Menues
  - Scroll bars
  - Dialoge . . .
- **Layout:** zur Anordnung von Komponenten innerhalb von Komponenten.
- **Events:** Ereignisse, zur Interaktion mit der Oberfläche notwendig.

## Erstellen einer GUI

1. Erzeugen der Komponenten
  - wie üblich mittels `new` (`Button quit = new Button("Quit")`),
  - passiert meist zu “Beginn” der umgebenden Komponente (in `init()`, im Konstruktor)
2. Hinzufügen der Komponenten in einen Behälter
  - führt oft zu einer “*Enthalten-in*”-Hierarchie: Container in Container . . .
  - Hinzufügen einer Komponente zu einem Container: `add`
3. Festlegen des **Layouts**: Verschiedene Layoutmanager
4. Handling von **Events** (Später)

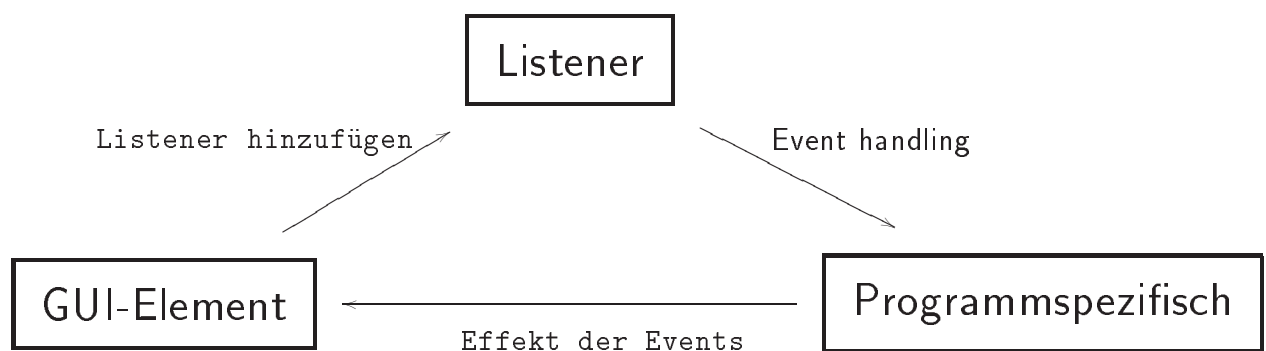


Abbildung 2: Events & GUIs

## GUI-Komponenten

Komponente	Nutzen
Component	Wurzel der Containerhierarchie
Button	Knopf zum Klicken mit String als Beschriftung.
Canvas	Vanilla-Oberfläche
Label	eine Zeile read-only Text
TextField	Zeile edierbaren Text (s.a. TextComponent-Oberklasse)
TextArea	Mehrere Zeilen edierbaren Text
List	List selektierbarer Elemente
Scrollbar	wie der Name schon sagt; senkrecht oder waagrecht
FileDialog	Dialogbox zum Filebrowsen
Checkbox	An/Aus (s.a. CheckboxGroup. "Checkbox/Radio buttons")
Choice	Auswahlmenu ("Choice-Button")

Tabelle 7: einige Komponenten (Unterklassen von Component)

## Komponenten (2)

Menu Component	Nutzen
Menu MenuBar MenuItem	Pulldown Menu, in einer Menueleiste; add( ) für Menuitems Menuleiste. add(Menu m) fügt neue Menues hinzu
Container	Nutzen
Container Dialog Window Frame Panel  ScrollPane	Wurzel der Containerhierarchie Dialogboxen Top-level Fenster ohne sonstigen Schmuck Fenster mit Rand (Unterklasse von Window Container innerhalb eines anderen, also nicht stand-alone. Generischer Container ohne eigenes Verhalten, hierarchisch schachtelbar, Oberklasse von Applet. einzelne Region mit zwei Scollbars, senkrecht und waagrecht

## Containers

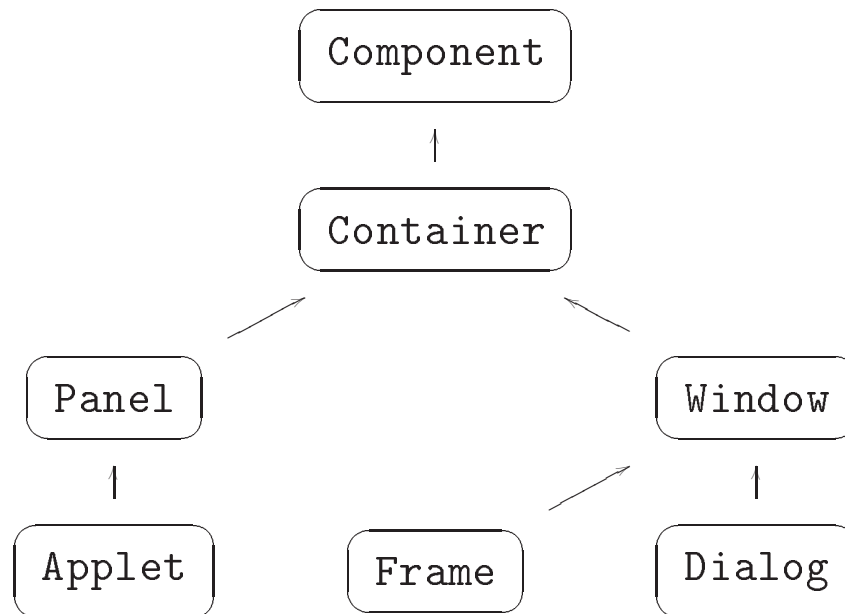


Abbildung 3: Containers

- spezielle Components zum **Gruppieren** von graph. Elementen
- zwei Arten von **Containern**
  - nur **innerhalb** anderer graphischer Container möglich: Panel/Applet (z.B. innerhalb des Appletviewers)
  - die **stand-alone** laufen können/eine eigene graphische Fläche im Fenstersystem bilden (siehe auch das Stand-Alone-Applet auf Folie 84).

---

## Layout Management

- `Layout Manager` implementieren `LayoutManager` interface
- Wichtige Gruppe von Klassen im `awt`
- Verwendung: Instanzen werden als Argument der Container-methode `setLayout(..)` gegeben.
- Neben dem default-Verhalten: keine Strategie (= `null` = Layout "per Hand") gibt es `vordefinierte` Manager für verschiedenen Strategien:

---

<code>FlowLayout</code>	Fließtextartig
<code>GridLayout</code>	Spalten- und Zeilenanordnung
<code>BorderLayout</code>	N/S/O/W und einen in der Mitte
<code>CardLayout</code>	Nur eine Komponente zu einem Zeitpunkt
<code>GridBagLayout</code>	komplexere Strategie

---



# Lektion IX

## Events

**Inhalt:** Java 1.1 Ereignismodell · ereignisgesteuerte Programme · Vergleich von Events in 1.0 und 1.1 · Listener-Interfaces · Adapterklassen

**Literatur:** Wir behandeln das Ereignismodell von Java 1.1. Wichtige Klassen finden sich in `java.awt.events` und auch `java.awt`. Weitere Information in Kapitel 7 von [Fla97b] sowie Kapitel 10 aus [LL97]. Wer bereits an *Swing* und dem Event Modell in Java 1.2 interessiert ist, findet in [Eck98] ein wenig Material, auch Diskussion der unterschiede der AWTs in 1.0, 1.1 und 1.2.

# Übersicht

- GUI-Interaktion: ereignisgesteuert  $\Rightarrow$  Kern der Gui-programmierung: **Events**
- **Eventmodell**: “Ereignisprotokoll”. Unterschiedlich in
  - Java 1.0: einfach, plump, objekt-desorientiert, für größere Oberflächen nicht geeignet, “mißbilligt” = *deprecated*
  - **Java 1.1**: komplexer, baut auf **innere Klassen**.<sup>36</sup>
  - Zukunft *Java 1.2*: **Swing** als GUI-Teil der *Java Foundation Classes* (JVC)
- Eventmodell: drei Mitspieler im Eventprotokoll
  1. **Event**:
    - Instanz der Klasse `java.util.EventObject`, bzw. deren Unterklasse `java.awt.AWTEvent`<sup>37</sup>
    - Quelle (`getSource`)
    - verschiedene “Sorten” von Events = verschiedene Unterklassen, Event-Klassenhierarchie
    - Definition eigener Eventklassen möglich
    - neue *Namenskonvention*, Zusammenarbeit mit JavaBeans
    - **Varianten** jeder Sorte als “Feld-Konstanten”

---

<sup>36</sup>Webbrowser unterstützen momentan meist nur das Ereignismodell von Java-1.0. Wir machen Version 1.1.

<sup>37</sup>in Java 1.0 war es die Klasse `Event`.

## 2. Quelle

- generiert Events
- benachrichtigt die (passenden) Listener durch **Methodenauf**ru $f$ , Event als Parameter
- Unterhält eine **Liste** von Listenern, die benachrichtigt sein wollen.
- erlaubt dynamische **An/Abmeldung** von Listener
- Beispiel für Ereignisquelle: `java.awt.Container`

## 3. Listener

- Empfänger der Ereignisse als Methodenparameter
- **implementiert Listener-Interfaces**
- für jede *Klasse* von Events: zugehöriges Interface<sup>38</sup>

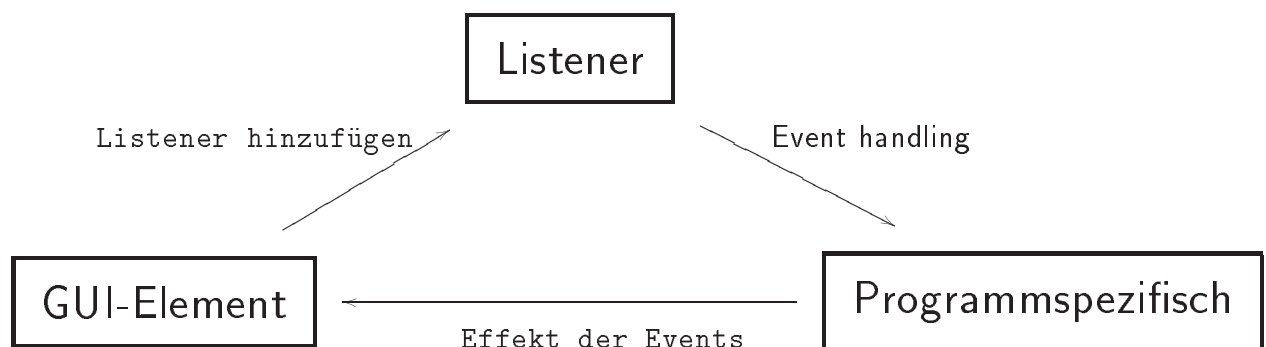


Abbildung 4: Events & GUIs

<sup>38</sup> Ausnahme: der Klasse `MouseEvent` sind 2 Listener-Schnittstellen zugeordnet: `MouseListener` und `MouseMotionListener`.

# Ereignismodell

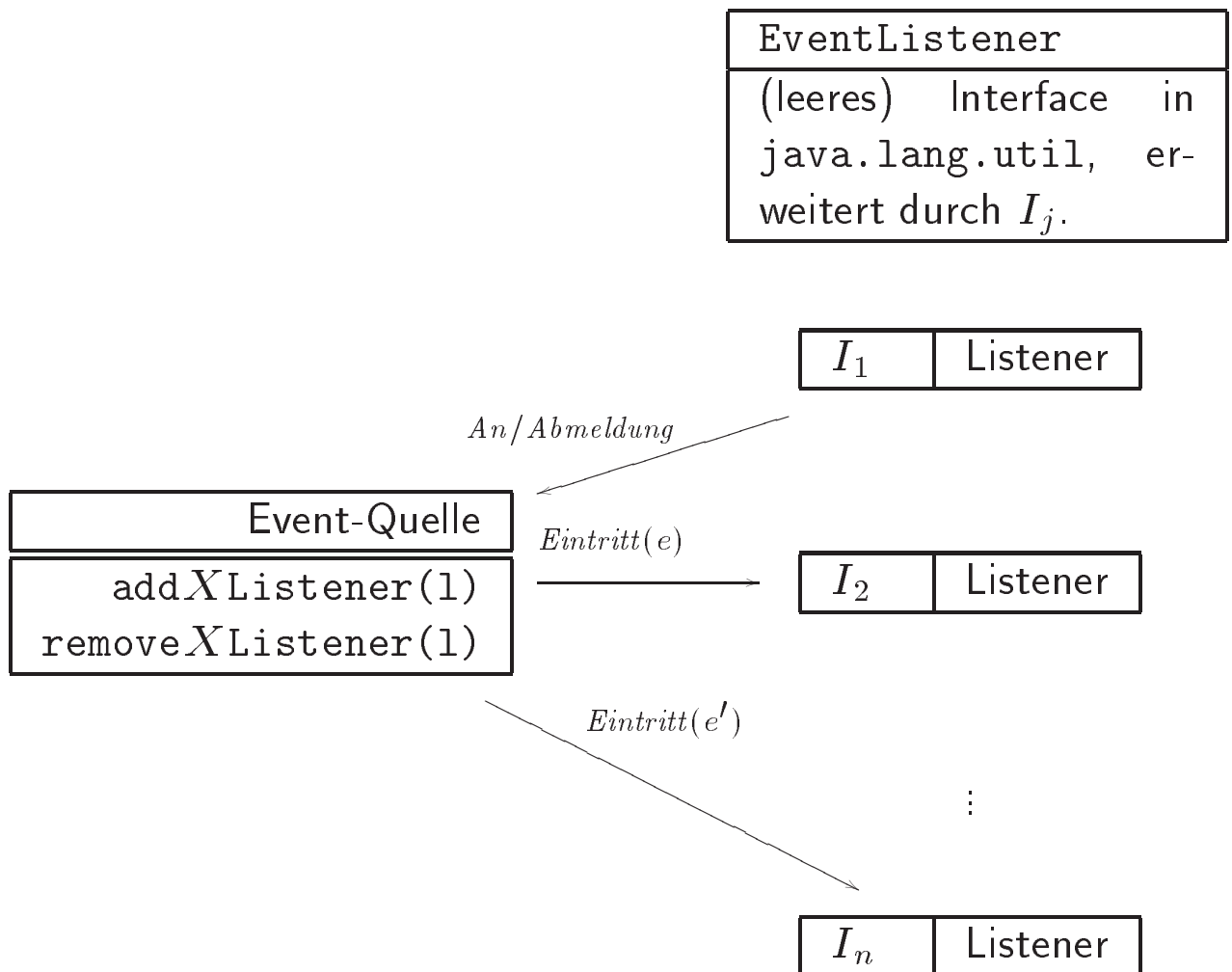


Abbildung 5: Event-Modell (AWT-Events)

---

## Java 1.0 gegen 1.1

**Beispiel 21 (Alt und Neu)** Folgendes kleine Applet stellt die Eventmodelle von Java 1.0 und 1.1 im Vergleich gegenüber. Man beachte die if-Kaskade im 1.0-Modell und die Verwendung innerer Klassen für 1.1

---

```
import java.awt.*;
import java.awt.event.*;          // Für Events
import java.applet.*;

public class AltundNeu extends Applet {
    Button          // Hier: 1.0 = 1.1
    b1 = new Button("Knopf 1"),
    b2 = new Button("Knopf 2"),
    b3 = new Button("Knopf 3(alt)"),
    b4 = new Button("Knopf 4(alt)");

    public void init(){          // Applet-Methode
        b1.addActionListener(new B1());
        b2.addActionListener(new B2());
        this.add(b1); add(b2); add(b3); add(b4);
    };

    // 1.1:      zwei innere klassen B1 und B2
    class B1 implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            getAppletContext().showStatus("Knopf 1");
        };
    };
};
```

```
class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        getAppletContext().showStatus("Knopf 2");
    };
};

// ----- Java 1.0 -----
public boolean action(Event evt, Object arg) {
    if (evt.target.equals(b3))
        getAppletContext().showStatus("Knopf 3");
    else if (evt.target.equals(b4))
        getAppletContext().showStatus("Knopf 4");
    else return super.action(evt, arg);
    return true;          // true als Erfolgsmeldung
};
};
```

---

---

## Adapter-Klassen

- zu jeder Sorte Event: ein entsprechendes **Interface**
- Interface muß durch den **Listener** implementiert werden ⇒ zwei Möglichkeiten
  1. **direkte** Implementierung des Interfaces mit `implements`
  2. **indirekt** mittels **Adapter**-Klasse
    - Adapter-Klasse: **triviale** Implementierung des zugehörigen Interfaces
    - Listener: durch Unterklassenbildung und **Überschreiben** der benötigten Methoden

## Eventklassen und Listenerklassen

- Namenskonvention: s. Table 8
- Beispiel 21 auf Folie 100: *Sorte* = Action
  - Quelle = Button erzeugt Instanzen von `ActionEvent.1`
  - innere Klasse B1 implementiert das Interface `ActionEventListener`.
  - Dies Interface listet auf, auf welche Methoden ein `ActionListener` horchen muß  
 ⇒ `actionPerformed` enthält den programmspezifischen Code
  - Registrierung des Listeners `new B1()` an der Quelle: `b2.addActionListener(new B1());`

<i>SorteEvent</i> <i>SorteListener</i> <i>SorteAdapter</i> <code>addSorteListener</code> <code>removeSorteListener</code>	Zu jeder <i>Sorte</i> (siehe nächste Folie) gibt es eine Klasse der Events und ein Interface der zugehörigen Listener sowie den Adapter. Daneben die Methoden in den Quellen zum An/Abmelder der Listener. Ausname: <i>Sorte</i> = Mouse (MouseListener und MouseMotionListener)
---	--

Tabelle 8: Eventklassen (Übersicht)



<i>Sorte</i>	Methoden	Quelle (nur die höchsten in der Hierarchie aufgelistet)
Action	actionPerformed	Button List TextField MenuItem
Adjustment	adjustmentValueChanged	Scrollbar
Container	componentAdded componentRemoved	Container
Component	componentMoved componentHidden componentResized componentShown	Component
Focus Item	focusGained, focusLost itemStateChanged	Component Checkbox, CheckboxMenuItem, Choice, List
Key	keyPressed keyReleased keyTyped	Component
Mouse	mouseClicked mouseEntered mouseExited mousePressed mouseReleased	Component
	mouseDragged MouseMoved	
Text	textValueChanged	TextComponent
Window	windowOpened WindowClosing windowActivated windowClosed windowDeactivated windowDeiconified windowIconified	Window

Tabelle 9: Eventklassen

# Lektion X

## Threads

**Inhalt:** Threads · Zustände von Threads · Synchronisation  
· Prioritäten · Threadgroups

**Literatur:** [Fla97b] gibt diesmal nicht soviel her, zumindest nicht auf einem Fleck konzentriert. In [LL97] werden Threads in Abschnitt 14.2 besprochen. Der entsprechende Abschnitt in Suns Javatutorial [CW96] ist ebenfalls recht brauchbar. Deneben gibt es natürlich noch [OW97] nur über Threads in Java, aber ich kenne es nicht selbst. In der Klassenbibliothek sind `java.lang.Thread` und `java.lang.ThreadGroup` die wichtigsten Klassen zum Thema.

## Allgemeines

- **Definition:**

**Thread:** ein sequentieller Kontrollfluß innerhalb eines Programmes

- bezeichnet auch als **lightweight processes**
- **quasi-parallele** Ausführung mehrerer sequentieller Kontrollfüsse (**Multi-Threaded**)
- Bisher: Ein (Anwender)-Programm/Applet = ein Thread (single threaded), aber:
- im **Laufzeitsystem**: Threads, die von der **Java Virtual Machine** automatisch erzeugt und verwaltet werden: **Dämonen**
  - Garbage Collection: niederpriorer Hintergrundthread
  - Threads für das Event-Handling
  - ein Thread bedient die `main`-Methode
- Programmterminierung: alle nicht-Dämonenthreads sind beendet
- **Thread**: Verwaltung von threads: starten, stoppen, Prioritäten setzen . . .
- Thread kann eine best. **Priorität** haben

## Threads in Java

- Klasse `java.lang.Thread`, muß also nicht importiert werden
- `Thread` implementiert `Runnable`
- zwei Arten, Threads zu generieren:

1. als `Unterklasse` der `Thread`-Klasse und überschreiben der `run`-Methode, oder
2. Implementierung des `Runnable`-Interface.

- `run()` ist der Rumpf der Threads
- Der `Thread`-Konstruktor nimmt die Implementierung des `Runnable`-Objektes als Argument<sup>39</sup>
- Ein neue Thread wird — wie üblich — mit `new` erzeugt und durch Aufruf von `start` gestartet.
- Thread endet, wenn das Ende der `run`-Methode erreicht wird.
- Mit `suspend` und `resume` kann die Verarbeitung eines threads angehalten/fortgesetzt werden.

---

<sup>39</sup>Thread hat verschiedene Konstruktoren.

**Beispiel 22 (Unterklasse von Thread)** Die Klasse erweitert einfach Threads und überschreibt die `run`-Methode.

---

```
public class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {                // "überschreiben
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
};
```

---

**Beispiel 23 (Implementieren von Runnable)** Dies ist die zweite Möglichkeit, wie man einen Thread bekommen kann. Der **Konstruktor** der Thread-Klasse macht aus einem Objekt, das ein Runnable-Interface implementiert, den eigentlichen Thread. Dieser muß dann extra **gestartet** werden. Das starten wiederum ruft **run** auf.

---

```
import java.awt.Graphics;
import java.util.*;
import java.text.DateFormat;
import java.applet.Applet;

public class Clock extends Applet implements Runnable {
    private Thread clockThread = null;

    public void start() {
        if (clockThread == null) {
            clockThread = new Thread(this, "Clock"); // Konstruktor Thread
            clockThread.start();
        }
    }

    public void run() { // wg. Interface runnable
        Thread myThread = Thread.currentThread();
        while (clockThread == myThread) {
            repaint();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e){}
        }
    }

    public void paint(Graphics g) {
```

```
    Calendar cal = Calendar.getInstance();
    Date date = cal.getTime();
    DateFormat dateFormatter = DateFormat.getTimeInstance();
    g.drawString(dateFormatter.format(date), 5, 10);
}

public void stop() {                                // Applets stop Methode
    clockThread = null;
}
};
```

---

---

## Zustände von Threads

- **neu**: bereits instantiiert, aber noch nicht gestartet<sup>1</sup>
- **ausführbar**
- **tot**: am Ende der `run`-Methode. Oder auch nach Aufruf von `stop`, wird aber in Java-1.2 mißbillig
- **blockiert/nicht ausführbar**. Mehrere Gründe
  - **schlafend**: feste Zeitspanne
  - **suspendiert**: keine Reaktion bis `resume` von außen aufgerufen wird.
  - **wartend**: keine Reaktion bis `notify`
  - Aufruf einer **synchronisierten** Methode an einem Objekt, welches gerade anderweitig beschäftigt (`locked`) ist
  - Warten auf Beendigung von I/O, beispielsweise ein **Strom**



## Lebenszyklus eines Threads

Hier eine Zusammenfassung der Zustände und Übergänge von Threads:

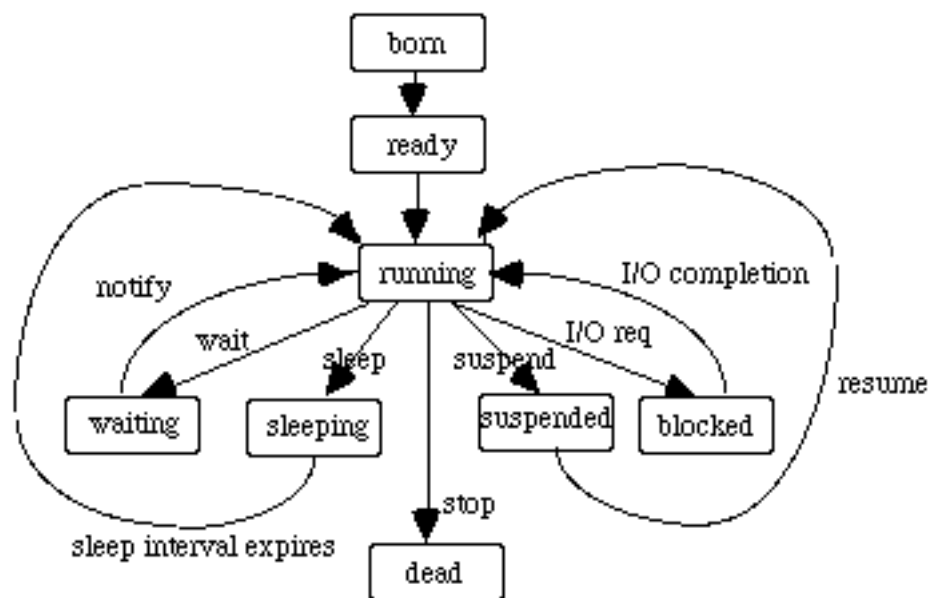


Abbildung 6: Zustände von Threads

## Synchronisation

- Quasi-paralleler Zugriff auf **gemeinsame Daten**  $\Rightarrow$

Synchronisation

notwendig

- **Kritischer Abschnitt**: Programmabschnitt, für den **gegenseitiger Ausschluß** also exklusiver Zugriff erforderlich ist.

**Beispiel 24 (Leser/Schreiber)** In folgendem Szenario haben die Objecte Schreiber und Leser Zugriff auf das selbe Speicherobject, der eine schreibend, der andere lesend. Das Speicherobject soll einfach eine **Speicherezelle** darstellen, auf die mit den Methoden **get** und **put** zugegriffen wird.



Es gibt zwei Dinge zu koordinieren:

- gegenseitiger Ausschluß: **locking**
- Unterschiedliche Geschwindigkeiten von Leser und Schreiber (**race conditions**): der Zugriff muß abgestimmt werden.

## Gegenseitiger Ausschluß (`synchronized`)

- zur Erzwingung gegenseitigen Ausschlüssen in kritischen Abschnitten: Schlüsselwort `synchronized`
- kann
  - Programm-Block
  - Methodeunter Ausschluß setzen
- assoziiert einen sog. `lock` mit der geschützten Programmeinheit
- zweierlei syntaktische Verwendung

### 1. als `Statement`

- Syntax:

```
synchronized (expression) statement
```

- `expression`: Schlüssel/Semaphore für den Zugriff (muß sich zu einem `Object` oder `Array` auswerten)
- `statement`: Rumpf des `kritischen Abschnittes`

### 2. als Methoden-Modifikator

- Syntax:

```
public synchronized typ name (parms) {  
    ....  
};
```

- Methodenrumpf als kritischen Abschnitt.
  - für **Klassenmethoden** (`static`): Lock auf die gesamte Klasse.
  - für **Instanzmethoden**: ein Lock auf die gesamte Instanz<sup>40</sup>
  - bevor eine synchronisierte Methode ausgeführt werden kann: Lock notwendig  $\Rightarrow$  nur je eine (synchronisierte) Methode kann an der Instanz bzw. der Klasse zu einer Zeit ausgeführt werden
- Aber: Locks erlauben “**Reentranten**” Kode

---

```
public class Reentrant {
    public synchronized void a() {
        b();
        System.out.println("here I am, in a()");
    }
    public synchronized void b() {
        System.out.println("here I am, in b()");
    }
}
```

---

---

<sup>40</sup> nicht nur auf die Methode.

**Beispiel 25 (Leser/Schreiber (2))** Da im Leser/Schreiber-Beispiel der Weg sein soll, auf den gemeinsamen Daten zu operieren, die beiden Zugriffsmethoden `get` und `put` sind, kann man die gleichzeitigen Zugriff dadurch erzwingen, daß man die Methoden als `synchronized` markiert.

```
public class Speicher {      // Speicher mu"s kein Thread sein
    private int speicherzelle // das gemeinsame Datum

    public synchronized int get () {
        ....
    };

    public synchronized void put (int value) {
        ....
    };
};
```

## Warten auf Nachricht

- nicht nur Verhindern von gleichzeitigen Zugriff, sondern auch explizites
  - **Warten** auf das Eintreten von Ereignissen (genauer eine Benachrichtigung) und
  - **Benachrichtigen** von deren Eintreten notwendig
- Instanzmethoden in `java.lang.Object`
  - **wait**: Warten auf Benachrichtigung
  - drei Formen, zwei mit **Timeout**
    - \* `wait()`
    - \* `wait (long timeout)`
    - \* `wait (long timeout, int nanoseconds)`
  - **notify**: benachrichtigen einer bestimmten Instanz.
  - **notifyAll**: alle benachrichtigen
- Vorsicht vor **Deadlocks** durch **zyklisches Warten**
- Die erwähnten Methoden sind nur **innerhalb** von synchronisierten Blocks oder Methoden erlaubt.
- wichtiger **Unterschied** zu **suspend/resume**: Im Wartezustand gibt der Thread den **Lock** frei

**Beispiel 26 (Leser/Schreiber (3))** In dem Beispiel mit dem Leser und dem Schreiber kann man diese Methoden verwenden, um die Geschwindigkeit von beiden Threads anzugleichen.

```
public synchronized int get() {
    while (available == false) { // Explizite Schleife
        try {
            wait(); // Warten auf den Schreiber
        } catch (InterruptedException e) {
        }
    }
    available = false;
    notifyAll(); // Benachrichtigen
    return contents;
}
```

```
public synchronized void put(int value) {
    while (available == true) {
        try {
            wait(); // Warten auf den Leser
        } catch (InterruptedException e) {
        }
    }
    contents = value;
    available = true;
    notifyAll(); // Benachrichtigen
}
```

## Sonstige Interaktion mit Threads

- `sleep(long milliseconds)`
  - für festgelegte Zeitspanne `deaktivieren`/pausieren
  - Klassenmethode der Klasse `Thread`
  - Verwendung oft einfach `Thread.sleep(100)`
- `join()`: Instanzmethode, warten auf den `Tod` eines Threads
- Mißbilligte Methoden (ab Java 1.2 *deprecated*)
  - `suspend/resume`: zweifaches Anhalten und Weitermachen
    - \* ein suspendiertes Objekt gibt (anders als ein *wartendes*) seine Locks `nicht frei` ⇒
    - \* führt leicht zu `Verklemmungen`
  - `stop`: Abbrechen
    - \* gibt alle Locks frei ⇒
    - \* Gefahr von `inkonsistentem Zustand`
    - \* wirft eine `Ausnahme`, und zwar eine Unterklasse von `Error` ⇒ fehlerhafter Abbruch
  - `interrupt`: unterbrechen
  - `destroy`: wurden (noch) nie implementiert, eine `suspend` ohne `resume`.<sup>41</sup>

---

<sup>41</sup>Streng genommen, keine *deprecated* Methode



## Alternative zum Suspendieren

**Beispiel 27 (Suspendieren)** Folgendes Beispiel zeigt, wie man die Methoden `suspend` und `resume` umgehen kann, indem man `wait` und `notify` und ein **Flag** (hier `suspended` verwendet. An Stelle von `suspend` und `resume` werden Methoden `mySuspend` und `myResume`. Das Flag schaltet den *suspend*-Zustand ein und aus. Das `wait` muß synchronisiert sein.

---

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class Suspend extends Applet {
    private TextField t = new TextField(10); // output
    private Button      // zwei Knöpfe
        suspendbutton = new Button("Suspend"),
        resumebutton  = new Button("Resume");

    public class Suspendable extends Thread {
        private int count =0;
        private boolean suspended = false; // Flag
        public Suspendable () { start();};

        public void mySuspend() {
            suspended = true; // Flag => true
        };
    };
};
```

```
public synchronized void myResume(){
    suspended = false;           // Flag => false
    notify();
};

public void run() {              // Runnable
    while (true) {
try {
    sleep(100);
    synchronized(this) {        // auf sich selbst
        while (suspended) wait(); // sync-Block
    }
} catch (InterruptedException e){};
t.setText(Integer.toString(count++));
    };
};

private Suspendable ss = new Suspendable();

public void init() {            // Applet
    add(t);

    suspendbutton.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
ss.mySuspend();
        }
    });
    resumebutton.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
ss.myResume();
        }
    });
};
```

```
    });
    add(suspendbutton);           // Knöpfe ans
    add(resumebutton);           // Applet heften
};                                // Ende init

public static void main(String[] args) { // stand-alone
    Suspend applet = new Suspend();
    Frame aFrame = new Frame("Suspend");
    aFrame.addWindowListener(new WindowAdapter(){
        public void windowClosing(WindowEvent e){
System.exit(0);                    // ordentlich
        };                          // terminieren
    });

    aFrame.add(applet, BorderLayout.CENTER);
    aFrame.setSize(300, 300);
    applet.init();
    applet.start();                // thread starten
    aFrame.setVisible(true);      // = show
};
};
```

## Prioritäten

- quasi-parallele Ausführung von mehreren Threads  $\Rightarrow$  Scheduler
- Javas *Laufzeitsystem*: Scheduling mit fester Priorität<sup>42</sup> (*fixed priority scheduling*)
  - mehrere Threads ausführbar: nimm den *dringlichsten*<sup>43</sup>
  - *unterbrechend*: es läuft *immer*<sup>44</sup> der ausführbare Thread mit der höchsten Priorität, solange bis er
    - \* endet<sup>45</sup>
    - \* ein Ergebnis liefert (*yield*) oder sonstwie
    - \* *nicht-laufend* (non-runnable) wird
    - \* ein dringlicherer Thread ausführbar wird.
  - Bei Gleichstand: *zyklische*, nicht-interbreachende Auswahl. (*round robin*)
  - Manche Betriebssysteme (nicht Javas Laufzeitsystem) erzwingen *Time-slicing*, aber: nicht darauf verlassen (Plattformunabhängigkeit)

---

<sup>42</sup>zwischen `Thread.MAX_PRIORITY` und `Thread.MIN_PRIORITY`. Default: `Thread.NORM_PRIORITY`.

<sup>43</sup>Je kleiner die Zahl, desto höher die Dringlichkeit oder Priorität.

<sup>44</sup>Fast immer. Der Fairness halber, um *Aushungern* einzelner Threads zu verhindern kann es Ausnahmen geben. In keinem Fall darf die Korrektheit eines Algorithmusses von Prioritäten oder der Schedulingstrategie abhängen.

<sup>45</sup>oder mit `stop` abgebrochen wird, was aber nicht empfohlen wird.

- Thread *erbt* die Priorität seines *Erzeugers*
- *Verändern* der Priorität: `setPriority`, Lesen mit `getPriority`
- Verhindern von *Aushungern*:
  - `yield`: *statische* Methode: Thread gibt die Kontrolle auf, läßt anderen *gleichprioren* Thread den Vortritt

---

## Gruppen von Threads

- Klasse `java.lang.ThreadGroup`.
- Jeder Thread gehört zu einer Gruppe (notfalls der System-Defaultgruppe)
- Gruppen von zusammengehörenden Threads
- **statische** Gruppenzugehörigkeit
- Operationen auf der Gruppe als Ganzem
  - Informationen über alle Threads der Gruppe sammeln
  - Gruppe selbst hat **Attribute** (z.B. maximale Priorität)
  - Interaktion mit allen **Threads** der Gruppe auf einmal (alle auf einmal starten etc.)
  - **Zugriffsbeschränkungen**, basierend auf Gruppenzugehörigkeit (`SecurityManager`)
- **Hierarchie** von Thread-Gruppen

# Lektion XI

## Networking

**Inhalt:** Internetadressierung · Sockets und Ports  
· Client/Server-Programmierung · Multicasting · höhere Netz-  
dienste · Remote Method Invocation

**Literatur:** Das relevante API Paket ist `java.net`; für *remote method invocation* `java.rmi.*`. Material und Beispiele zum Thema findet sich in Kapitel 9 von [Fla97a] sowie in Kapitel 15 von [Eck98]. Auch der entsprechende Abschnitt aus dem Tutorial [CW96] ist als Einstieg brauchbar. Darüberhinaus gibt es noch [Har97] nur über Netzwerkprogrammierung, aber ich kenne es selbst nicht. Ein wenig über die verschiedenen Internetprotokolle habe ich aus [Tan96].

*Remote Method Invocation* wäre eigentlich ein Kapitel für sich. Literatur dazu findet sich im Nutshellbuch [CFM98] über

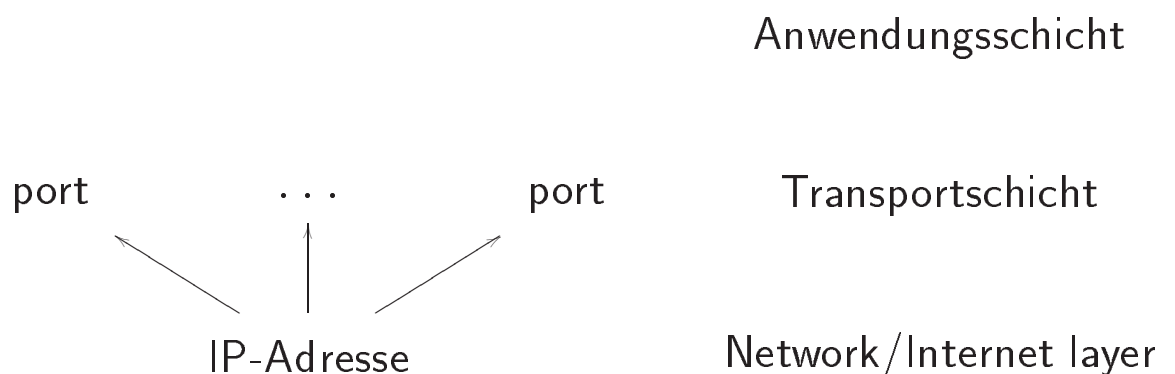
---

Javas *Enterprise*-Klasse. Das Buch ist allerdings noch nicht erhältlich ist. Einiges ist anscheinend auch in [Har97] enthalten



## IP-Adressen & Ports

- **IP-Adressen:** Adressierungsschema des *Internet-Protokolls*, momentan (IPv4) *32 Bit*
  - in dezimaler Punkt-Schreibweise 134.245.253.7
  - **DNS-Namen:** garfield.informatik.uni-kiel.de
  - Spezialadressen, z.B. 127.xx.yy.zz: **Loopback**, gut zum Testen
- **Port**
  - TCP-Schichten Modell
  - Transport Layer Service Access Point, d.h., dort stehen **Transportschicht-Dienste** zur Verfügung
  - 16-bit Port Nummer.  $\leq 256$ : “wohlbekannte“ Ports<sup>46</sup>, bis 1024 reserviert



<sup>46</sup> z.B. 21 = FTP, 23 = Telnet, etc. Siehe RFC 1700

**Beispiel 28 (Adresse)** Mittels *statischer* Methoden der Klasse `InetAddress` kann man die TCP/IP-DNS-Adreßdienste nutzen. Instanzen von `InetAddress` sind IP-Adressen (allerdings hat die Klasse keinen Konstruktor).

---

```
import java.net.*;

public class NSLookup { // einfache DNS-Anfrage
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.err.println("Usage: NSLookup machine");
            System.exit(0); //
        };
        InetAddress la = InetAddress.getLocalHost();
        InetAddress a = InetAddress.getByName(args[0]);
        System.out.println(a);
        System.out.println("Lokaler Host: " + la);
    };
};
```

## Sockets

- **Sockets**: Endpunkte einer Transportverbindung
- dienen zur Interprozesskommunikation
- Adressierung der Sockets:

Socket = IP-Adresse × Port-Nummer

d.h., Socket = Transportschichtadresse, in Java als Instanzen der Klasse `java.net.Socket`

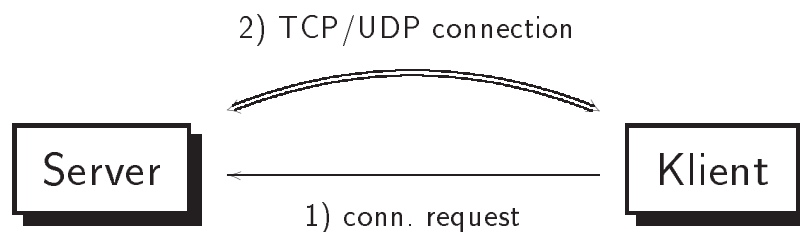
- Mehrere Verbindungen können über den selben Port/Socket laufen
- zwei Transportprotokolle:<sup>47</sup>, beide Ende-zu-Ende-Protokolle, beide voll-duplex
  1. Verbindungsorientiert: **TCP**
    - fehlergesichert
    - reihenfolgegesichert ⇒ **Bytestrom**
    - Klasse `Socket`
  2. verbindungslos: **UDP**
    - **Datagram-Dienst**
    - Datagramme als Instanzen von `DatagramPacket`
    - Klasse `DatagramSocket`

---

<sup>47</sup>TCP und UDP sind die zwei Internettransportprotokolle *transmission control protocol* und *user data protocol*.

## Client/Servers

- **Server** stellt Dienste zur Verfügung; im OO-Modell als Methoden
- **Klienten** fordern Dienste an; im OO-Modell durch Methodenaufruf



- Der Aufbau der Verbindung geschieht in zwei Phasen
  1. Klasse `ServerSocket`: **Server** eröffnet den Socket auf dem er nach Verbindungswünschen von Klienten **lauscht**. Dieser Port muß allgemein bekannt sein  $\Rightarrow$  Methode `accept()`: **Server wartet** auf Klienten
  2. Klient meldet sich am Port  $\Rightarrow$  Aufruf von `accept` **deblockiert**<sup>48</sup> und liefert den **neuen** Socket für die Kommunikation als Ergebnis

<sup>48</sup>vergleiche das Kapitel über Threads

```
ServerSocket s = new ServerSocket(PORT);
try {
    Socket socket = s.accept();
    ....
}
finally {
    socket.close();
}
```

- Schließen mit `close()`
- Kommunikation über Sockets: mittels Strömen, genau wie sequentielles File-I/O.
- Duplex-Verbindung = bidirektional, d.h., ein Socket entspricht zwei Byteströmen (s. Lektion I/O)
  - Eingehender Strom (InputStream): Methode `getInputStream()`
  - Ausgehender Strom (OutputStream): Methode `getOutputStream()`
- Umwandlung in gepufferte Zeichenströme: mittels
  - `InputStreamReader` und `BufferedReader`, bzw.
  - `OutputStreamWriter` und `BufferedWriter`, wenn man wie üblich<sup>49</sup> mit `print` arbeiten will, auch noch `PrintWriter`

---

<sup>49</sup>Wie z.B. mittels `System.out`

**Beispiel 29 (Server)** Dies ist ein einfacher Server, der einen Socket aufmacht und zeilenweise Zeichen liest.

---

```
import java.io.*;
import java.net.*;
public class SingleServer {
    public static final int PORT = 2000;

    public static void main (String[] args) throws IOException {
        ServerSocket s = new ServerSocket(PORT);
        System.out.println("Started: " + s);
        try {
            Socket socket = s.accept();
            try {
                System.out.println("Connection accepted:" + socket);
                BufferedReader in =
                    new BufferedReader(
                        new InputStreamReader(socket.getInputStream()));

                PrintWriter out =
                    new PrintWriter(
                        new BufferedWriter(
                            new OutputStreamWriter(socket.getOutputStream())),
                        true);

                while (true) { // lesen bis "END"
                    String str = in.readLine();
                    if (str.equals("END")) break;
                    System.out.println("Echoing: " + str);
                    out.println("Echoing: " + str);
                }
            }
        }
    }
}
```

```
    }  
    finally {  
        System.out.println("closing ..");  
        socket.close();  
    }  
    finally {s.close();}};  
};
```

---

**Beispiel 30 (Client)** Hier der zu dem Server aus Beispiel 29 auf Folie 133 passende Klient.

---

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

public class SingleClient {
    public static void main (String[] args) throws IOException {
        InetAddress addr = InetAddress.getByName(null); // local host
        System.out.println("addr = " + addr);

        Socket socket = new Socket (addr, SingleServer.PORT);
        try {
            System.out.println("socket = " + socket);
            BufferedReader in =
                new BufferedReader(
                    new InputStreamReader (socket.getInputStream()));
            PrintWriter out =
                new PrintWriter(
                    new BufferedWriter(
                        new OutputStreamWriter(socket.getOutputStream())),
                    true);

            for (int i = 0; i < 10; i++){
                out.println("Hello World" + i);
                String str = in.readLine();
                System.out.println(str);
            };
            out.println("END");
        } finally {
            System.out.println("closing...");
        }
    }
}
```



```
        socket.close();  
    }  
};  
};
```

---

## Mehrere Klienten

- in der Regel: ein Server für **dynamische** Anzahl von Klienten
- Server aus Beispiel 29 kann nur einen Klienten **gleichzeitig** bedienen
- Abhilfe: **Threads**
  - der Multi-Server behält den vereinbarten Port im Auge
  - sobald neue Anfrage: **Erzeugung** ("spawn") eines neuen Server-Threads
  - Socket als Konstruktor-**Parameter**

**Beispiel 31 (Server)** Hier der entsprechende Code. Der

---

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

public class MultiServer {           // Startet die Unterthreads
    public static final int PORT = 2000;
    public static void main(String[] args)
        throws IOException {
        ServerSocket s = new ServerSocket(PORT); // Zum Horchen
        System.out.println("Started: " + s);
        try {
            while (true) {           // Soviele wie verlangt
                Socket socket = s.accept();
                try {
                    new ServeOne(socket); // socket als Param.
                }
            }
        }
    }
}
```

```
        } catch (IOException e) {
            socket.close();
        };
    };
}
finally {s.close();};
};
};
```

---

**Beispiel 32 (Server)** Hier eine einzelner Serverprozeß. Er bekommt den `Socket` den er bedienen soll, als Parameter.

---

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

public class ServeOne extends Thread {
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;

    public ServeOne (Socket s) throws IOException {
        // Socket als Parameter
        socket = s;
        BufferedReader in = // genau vor vorher
            new BufferedReader(
                new InputStreamReader(socket.getInputStream()));

        PrintWriter out = // genau wie vorher
```

```
        new PrintWriter(  
            new BufferedWriter(  
                new OutputStreamWriter(socket.getOutputStream()),  
                true);  
  
    start();                // ruft run() auf  
};  
  
public void run () {      // "Uberschreiben der run-Methode  
    try {  
        while (true) {    // Loop  
            String str = in.readLine();  
            if (str.equals("END")) break;  
            System.out.println("Echoing: " + str);  
            out.println(str);  
        };  
  
        System.out.println("closing ..");  
    } catch (IOException e) {  
    } finally {  
        try {  
            socket.close();  
        } catch (IOException e) {}  
    };  
};  
};
```

---

**Beispiel 33 (Klienten)** Der Vollständigkeit halber noch das selbe für Klientenseite. Hier der Prozeß, der alle Klienten erzeugt:

---

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

public class ManyClients {
    static final int MAX_THREADS = 4;
    public static void main(String[] args)
        throws IOException, InterruptedException {
        InetAddress addr = InetAddress.getByName(null); //
        while (true) {
            if (OneClient.threadCount() < MAX_THREADS)
new OneClient(addr);
            Thread.currentThread().sleep(1000);
        };
    };
};
```

---

**Beispiel 34 (ein Klient)** Schließlich ein einzelner Klient, aber das ist nicht wesentlich anders als zuvor:

---

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

public class OneClient extends Thread{
```

---

```
private Socket socket;
private BufferedReader in;
private PrintWriter out;
private static int counter = 0;
private int id = counter++;
private static int threadcount = 0;
public static int threadCount() {
    return threadcount;
};

public OneClient(InetAddress addr) {
    System.out.println("Neuer Klient @ " + id);
    threadcount++;
    try {
        socket = new Socket(addr, MultiServer.PORT); //stat. Adresse
    } catch (IOException e) {}
    try {
        in =
            new BufferedReader(
                new InputStreamReader (socket.getInputStream()));
        out =
            new PrintWriter(
                new BufferedWriter(
                    new OutputStreamWriter(socket.getOutputStream())),
                true);

        start(); // starte Thread -> run
    } catch (IOException e) {
        try {
            socket.close();
        } catch (IOException e2) {}
    }
};
```

```
};

public void run() {
    try{
        for (int i = 0; i < 25; i++){
            out.println("Client" + id + ": " + i );
            String str = in.readLine();
            System.out.println(str);
        };
        out.println("END");
    } catch (IOException e) {
    } finally {
        try {
            socket.close();
        } catch (IOException e) {}
        threadcount--;
    };
};
};
```

---

e

## Datagrammdienste

- funktioniert ähnlich wie ServerSockets nur mit DatagramSocket.
- basierend auf UDP
- Klasse Datagramm: Instanzen sind, klar, Datagramme, also UDP-SDUs:
- zwei Konstruktoren: zum Empfangen und zum Senden
  - `DatagramPacket(buf, buf.length)`
  - `DatagramPacket(buf, buf.length, inetAddress, port)`
- Gesendet werden: Arrays von Bytes (Maximum < 64KByte)
- verbindungsloser Dienst ⇒ Es gibt *keine Analogie* zu ServerSocket
- Codefragmente:
  - Empfangen:

```
DatagramPacket dp = new DatagramPacket(buf, buf.length);
...
s.receive(dp);                // s = datagram-socket
....dp.getAddress();
....dp.getPort();
```
  - Senden: analog, mit dem zweiten Konstruktor
- Spezialfall: Multicasting



## Multicasting

- **Multicasting** über Klasse `MulticastSocket`
- funktioniert ähnlich dem `Datagrammdienst`, andere Adressierung
- `MulticastSocket` ist Unterklasse von `DatagramSocket`
- `MulticastSocket` = selektiver `Datagramm-Empfangssocket`
- Senden/Empfangen von `Datagrammen` in einer `Gruppe` von Empfängern.
- zwei Instanzmethoden (= zwei `Protokolldienste`) eines `Multicastsockets` zur Gruppenverwaltung:
  - `joinGroup(InetAddress multicastaddress)`: Klient ruft diese Methode auf und wird `Gruppenmitglied`.
  - `leaveGroup(InetAddress multicastaddress)`: analog
  - reservierte Adressen 224.0.0.1 – 239.255.255.255<sup>50</sup>
  - Gruppen sind `offen`: Zum *Senden* an eine Gruppe muß man *kein* Mitglied sein, zum Empfangen schon
- Entscheidend ist
  - der `Datagramm-Socket` als `Multicast-Socket` beim `Empfänger`
  - Art des `Datagramms`: nämlich mit der `Multicastadresse`

---

<sup>50</sup>Zur momentanen IP-Adressierungsschema s. [Tan96].

- Codefragment:

```
MulticastSocket socket =  
    new MulticastSocket(4532);           // Port-Nr.  
InetAddress group =  
    InetAddress.getByName("230.0.0.1"); // Multicast-Addr.  
socket.joinGroup(group);
```

## Höhere Dienste: URL

- **URL**: *Uniform resource allocator*
- eindeutige **Adressierung** von Netzressourcen (oft HTML-Seiten oder Code):
- **Format** von URLs. Die Adresse wird meist als DNS-Name angegeben. Bei dem Dateipfad steht ein / am Ende für die Standarddatei `index.html`.

Bestandteil		Methoden
Protokoll/Scheme		<code>getProtocol</code>
Resource-Adresse	Host-IP-Adresse	<code>getHost</code>
	Datei (inkl. Pfad)	<code>getFile</code>
	Portnummer (opt.)	<code>getPort</code>
	Referenz (opt.)	<code>getRef</code>

- URL's als Java-Objekte. Beispiel:<sup>51</sup>

```
String hostname = "www.informatik.uni-kiel.de";
String protocol = "http";
String filename = "/inf/deRoeever/SS98/Java/"; // = index.html
URL kursurl    = new URL(protocol,
                        hostname,
                        filename);
```

<sup>51</sup>Man hätte auch den String im Ganzen übergeben können: `URL(protocol + "://" + hostname + filename)`.

- **relative** Urls in Java: mittels des überladenen `Url`-Konstruktors:

```
URL (URL baseURL, String relativeURL)
```

Zum Beispiel:

```
URL gruppenitem_url = new URL(kursurl, "#gruppen");  
URL gruppen_url     = new URL(kursurl, "gruppen.html");
```

- Instanzen von `URL` sind **unveränderliche** Objekte, d.h., es gibt keine Methoden `setFile` etc.

## URL: Zusammenfassung

Protokoll	Verwendung	Beispiel
http	Hypertext (HTML)	<code>http://www.informatik.uni-kiel.de/~ms</code>
ftp	File transfer	<code>http://www.informatik.uni-kiel.de/pub/README</code>
file	lokale Datei	<code>/home/ms/.plan</code>
news	Brett/Artikel	<code>news:cau.ifi.fragen</code> <code>news:AA4534345@news.informatik.uni-kiel.de</code>
gopher	Gopher	<code>gopher://gopher.informatik.tu-muenchen.de/</code>
mailto	Email	<code>mailto:ms@informatik.uni-kiel.de</code>
telnet	Einloggen	<code>telnet://snoopy.informatik.uni-kiel.de:80</code>

Tabelle 10: Verschiedene URL's

## Url-Verbindungen

- Kommunikation mittels Url: **öffnen** und **schließen** passender **Ströme**<sup>52</sup>

Methode	Rückgabetyt	Nutzen
<code>openConnection()</code>	<code>URLConnection</code>	zum Lesen und Schreiben von und auf die <code>URLConnection</code> .
<code>openStream()</code>	<code>InputStream</code>	erlaubt das <b>direkte</b> Lesen von einem Url,

- **Verbindungen** auf Anwendungsschicht über Urls
  - Instanzen von `URLConnection`
  - Ergebnis von `openConnection()`, kein Konstruktor<sup>53</sup>
  - Erlaubt mehr Kontrolle über die Verbindung also die `openStream`-Methode eines Url-Objektes
  - Lesen und Schreiben möglich, aber default: **nur Lesen**. Manipulation mit `setDoInput` und `setDoOutput`
  - eine Reihe weiterer Methoden, bitte selber nachschlagen

<sup>52</sup>Das ganze ist analog dem Vorgehen bei Sockets und Ports oder normalem File-I/O. Ports gehören zur *Transportschicht*, während URLs zur *Anwendungsschicht* gehören. Ebenfalls zur Anwendungsschicht gehören die Protokolle ftp, mail, telnet, http, etc.

<sup>53</sup>Entspricht ungefähr `accept` bei `ServerSocket` auf Transportschichtebene.

## Remote Method Invocation

- bisher: richtig *verteilte Berechnungen* Anwendungen — bis auf das low-Level Client/Server-Modell — haben wir noch nicht kennengelernt, nur
  - *verteilten Code* (*distributed code*)  $\neq$  verteilte Programmierung mittels Applets
  - (hauptsächlich) Lesen *verteilter Daten* durch das Laden von File-Urls
  - *quasi-parallele* Ausführung durch mehrere Threads
- echte Verteilung:

Methodenaufruf an Objekten auf anderen Maschinen

- Pakete ( $\geq$  Java 1.1)

java.rmi	Remote interface und Exception
java.rmi.server	u.a. remote Objects
java.rmi.dgc	Distributed GC
java.rmi.registry	Verwaltung von RO's

- gehört zu den *Enterprise*-Klassen

---

## RMI-Überblick

- Dies kann nur ein Einblick sein, RMI ist einigermaßen komplex
- Im Folgenden: **Server**: (Java)programm, welches Methoden über's Netz anbietet, **Klient** die Aufrufe.
- ein klein wenig wie die Client/Server auf den Sockets, aber
- wichtigste neue Zutat: Aufruf von Methoden an Objekten mit ihrem **Namen** anstelle von Kommunikation mit Ports
- **Bestandteile**:
  - **Remote Interface**: legt die von Ferne zugreifbaren Methoden fest
  - **Server**: implementiert das Interface
  - **Nameservice**: `rmiregistry`
  - **Klient**: erkundigt sich beim Nameservice nach den Servern



## RMI-Überblick II

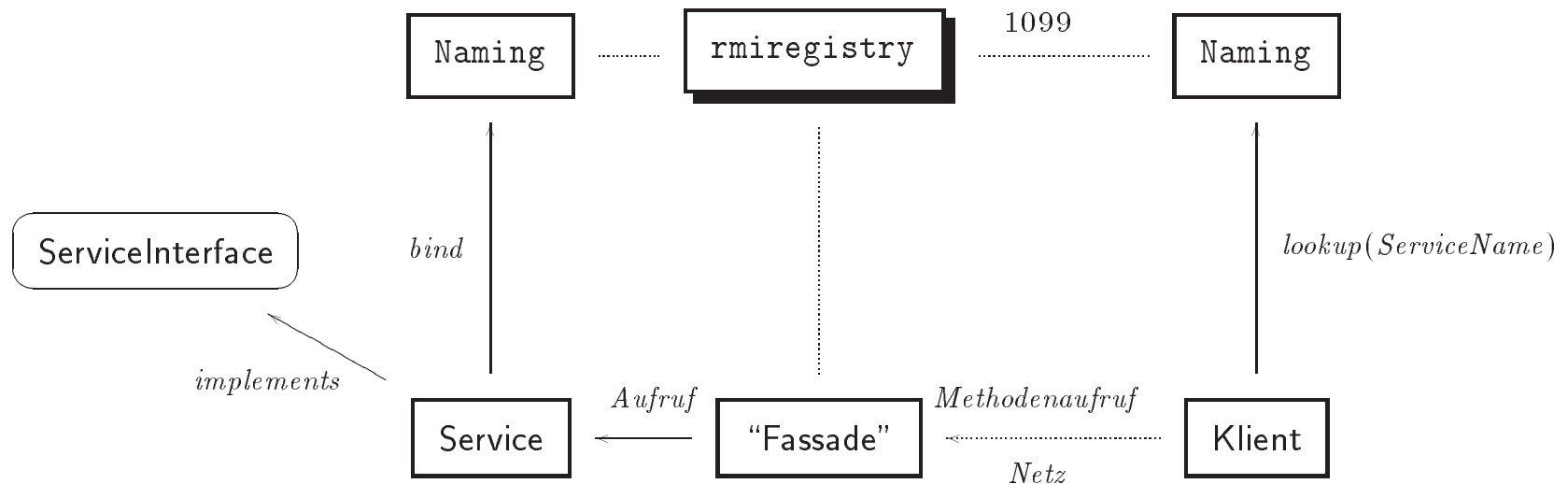


Abbildung 7: Remote method invocation

## Remote Interfaces

- **remote interface**
  - Erweiterungen des (leeren) Interface `java.rmi.Remote`<sup>54</sup>  
⇒ nur die Methoden die in einem remote Interface spezifiziert sind, sind auch von Ferne zugreifbar.
  - alle **Methoden** des Interfaces: throws `RemoteException` notwendig
  - die Schnittstelle muß `public` sein
  - die **Remote Objects**: deklariert als Implementierungen der remote interfaces (nicht ihrer Implementierungsklasse!)<sup>55</sup>
- **Implementierung eines RI**:
  - Klasse, deren Instanzen die verteilten Methoden zur Verfügung stellen
  - (meist) Erweiterungen von `UnicastRemoteObject`
  - Sicherheitsregistratur `java.rmi.RMISecurityManager`

---

<sup>54</sup>das einzige Interface aus `java.rmi.*`. Der Rest sind der Klassen dient fast nur der Ausnahmebehandlung.

<sup>55</sup>deswegen sind auch nur die Methoden, die im Interface genannt sind, von Ferne zugreifbar.

## Registration von Remote-Objekten

- separater Serverprozeß (= außerhalb des Interpreters)
  - Start mit `rmiregistry &`: Dämonprozeß<sup>56</sup>
  - `rmiregistry` läuft **unabhängig** vom `Javacompiler/Javaprogrammen`, also Obacht beim Starten innerhalb Javas (`LocateRegistry.createRegistry()`)
- Klasse `java.rmi.Naming`: dient zur Kommunikation mit dem Registrationsdämon und zur Namensverwaltung
- Beispiel:

```
Naming.bind("//snoopy/ServiceName", instanz);
```

- `snoopy` = Rechner,
- `ServiceName`: Name (String) unter dem der `Service` bekannt ist, es muß **nicht** der Name der Klasse sein
- `instanz`:
  - \* Instanz des `UnicastRemoteObject` und
  - \* Implementierung des `RemoteInterface`
- `rebind()`: dasselbe, aber es **ersetzt** ggf. einen Service
- **Lebenszeit** der Dienste:
  - solange `rmiregistry` läuft
  - bis `Naming.unbind()`

---

<sup>56</sup>überwacht i.d.R. Port 1099

---

## Programmstummel (Stubs) und Skelette

- Dem Server muß noch vorgespiegelt werden, daß er lokal aufgerufen wird ⇒
- Programm `rmic`: fügt dem kompilierten Code (hier Server) zwei neue Klassen hinzu:

```
Server_Stub.class  
Server_Skel.class
```

- Stub und Skeleton dienen als Fassade für das RO.

---

## Verwenden einen RO: Klienten

- Im Gegensatz zum Server: Klientenseite ist einfach
- Lokalisieren des Objektes mittels des rmiregistry-Dämons und `Naming.lookup`:

```
ServerInterface ro =                // Interface, nicht der Server
    (ServerInterface)Naming.lookup("//snoopy/ServiceName");
...
ro.method(...);                    // Methode des Interfaces
```

## Lektion XII

### Misc, Datastrukturen, Rekursion, Patterns etcetc.

**Literatur:** Der ganze Hype um Design Patterns startete mit dem Buch der "Viererbande": [GHJV95]. Siehe auch Kapitel 16 in [Eck98]. Ein kleines, nettes Buch, was einige wenige Patterns in Java einübt ist [FF98].

---

# Iterators

---

# Visitor



---

## Double Dispatch

- zugrundeliegende “Datentyp” bleibt unverändert,
- “double dispatching”

---

## Literatur

### Literatur

- [Bud97] Timothy Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley, 2 edition, 1997.
- [CFM98] William Crawford, Jim Farley, and Kris Magnusson. *Java Enterprise in a Nutshell, A Desktop Quick Reference*. O'Reilly, 1 edition, October 1998. To appear.
- [CW96] Mary Campione and Kathy Walrath. *The Java Tutorial*. The Java series. Addison-Wesley, 1996.
- [Eck98] Bruce Eckel. *Thinking in Java*. Prentice Hall, 1998.
- [FF98] Matthias Felleisen and Daniel P. Friedman. *A Little Java, A Few Patterns*. MIT Press, 1998.
- [Fla97a] David Flanagan. *Java Examples in a Nutshell*. O'Reilly, 1 edition, September 1997.
- [Fla97b] David Flanagan. *Java in a Nutshell*. O'Reilly, 2 edition, May 1997.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

- 
- [Har97] Eliotte Rusty Harold. *Java Network Programming*. O'Reilly, 1 edition, February 1997.
- [LL97] Lohn Lewis and William Loftus. *Java: Software Solutions*. Addison-Wesley, 1997.
- [OW97] Scott Oaks and Henry Wong. *Java Threads*. O'Reilly, 1 edition, January 1997.
- [Tan96] Andrew S. Tanenbaum. *Computer Networks*. International Series. Prentice Hall, third edition, 1996.