# Einführung in Java

Karsten Stahl & Martin Steffen

Sommersemester 2000



# Lektion I

## Einführung

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

**Literatur:** Aus den Nutshellbuch [Fla99b], dazu verschiedene Readmes, verschiedene Webpages (cf. die Kursseite

www.informatik.uni-kiel.de/inf/deRoever/SS00/Java/



## 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 Auftag von Time-Warner an Land ziehen)
- **April 93:** NSCA Mosaic verfügbar
- **Juni 93:** Deal mit Time-Warner kippt, SGI bekommt den Zuschlag
- **Sommer 93 – Frühjar 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 offiziell auf der *Sun-World '95* vor, Netscape springt auf und will Java lizensieren
- **später im Jahr:** weitere Firmen, darunter Microsoft, lizensieren 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 6.12 aus `comp.lang.java`)[1]
- Typinferenz
- type casts
- automatische Speicherverwaltung (garbage collection) : Lisp, 60ern

[1] Java 1.2 hat es auch noch nicht. Allerdings die Erweiterungen Generic Java und Pizza.





## Programmierumgebung

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

### 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 genaueres zur objektorientierte Programmstrukturierung eingehen, für's erste soll folgende Charakterisierung genügen:

- Klasse
  - Progammcode 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 Klasse
  - 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 automatisch in jedes Javaprogramm importiert. |
| `java.applet` | Programmierung von *Applets* |
| `jawa.awt` | Graphikunterstützung, GUIs |
| `java.beans` | wiederverwendbare SW-Komponenten |
| `java.io` | Input/Output |
| `java.math` | mathematische Funktionen |
| `java.net` | Netzprogrammierung |
| `java.rmi` | *Remote method invocation* |
| `java.security` | (Netzwerk-) Sicherheit |
| `java.sql` | Arbeiten mit Datenbanken |
| `java.text` | Textformatierung |
| `java.util` | Sonstige nützliche Klassen (Datentypen) |
| ... | etc. |

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:

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

- 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 {
    BuffererReader 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 ≠ 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 Schnelleinsstieg bietet *Java in a Nutshell* [Fla99b]. Der folgende Abschnitt baut zu einem Teil auf diesem Buch auf (Kapitel 2). Wer es gemächlicher und lehrbuchartiger wünscht, ist mit dem *Java Software Solutions* [LL97] gut bedient. In Bezug auf die Spezifika von Java 1.2 ist es ein wenig veraltet.



## Lexikalische Struktur

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

| | |
|---|---|
| `// Rest der Zeile ist Kommentar` | wie in $C^{++}$ |
| `/* <Kommentar> */` | wie in $C/C^{++}$ |
| `/** Dokumentarkommentar */` | für `javadoc` |

- Unicode-Zeichen[2] für Identifier, Strings
- kein Präprozessor (also kein `#define <KONST>`, `#include <File>`, `#ifdef`)
- String für Zeichenketten ist kein primitiver Datentyp, aber man kann Stringliterale wie gewohnt (`"Hello World"`) verwenden

[2] 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[3]
- 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`

[3] aber sich darauf zu verlassen, ist schlechter Stil.

| | Werte | Schlüsselwort | Bitlänge |
|---|---|---|---|
| numerisch | ganze Zahlen, alle *mit* 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[4]
- anonsten C-ähnlich (siehe ausgeteiltes Blatt), zusätzlich:
  - `>>>`: Rechtsschift mit 0
  - `|` und `&` auf `boolean`: *logisches* *„oder"* bzw. *„und"*[5]
  - `instanceof`
- es fehlen Operationen für Pointerarithmetik `*` (Dereferenzierung), `&` \\

[4] Der Operator ? beipiesweise für bedingte Ausdrucke (`<cond>?<expr1> : <expr2>`) ist dreistellig und *Mixfix*, Inkrement mit `++` ist einstellig und *Postfix*.

[5] Ansonsten *bitweise*.





## Referenzdatentypen

- Referenzdatentypen: Objekte und Arrays
- Manipulation "by reference", also über die Addresse/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
- Müllabfuhr (*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 elementaren 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[6]

[6] 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:[7]

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

- Bound-check zur Laufzeit
- Instantiieren 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}};
```

[7] Alternativ ist auch 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 Stringliterale wie gewohnt
  - Operator (keine Methode) + zur Konkatenation von Strings
- implizite Konversion in einen String in Fällen[8] wie:

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

[8] 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 | `if (<cond>) <stmnt1> else <stmnt2>` |
| switch | `switch (expr) { case value1: stmnt-list; case value2: stmnt-list; .... };` |
| | Schleifen |
| while | `while (<cond>) <stmnt>;` |
| for | `for (<init>; <cond>; <incr>) <stmnt>;` |
| do | `do <stmnt> while (<cond>);` |

- 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[9]/des Switchzweiges
- `continue`: Überspringen des weiteren Schleifenrumpfes

[9] `break` in Schleifen wird manchmal als schlechter Stil angesehen.





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

10. April 2000

# Lektion III

# Objektorientierte Strukturen (1)

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

**Literatur:** Die Lektion baut teilweise auf Kapitel 3 aus [Fla99b] 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`).[10]

[10] Das gleiche gilt für eine Methode ohne `return`.





**Beispiel 4 (Circle)** Der *Zustand* eines Kreises als einer Instanz von `Circle` — die Koordinaten seine 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`[11]
- Methoden/Feld-Selektion ("Methodenaufruf): mittels "`.`".

**Beispiel 5 (Circle)** Zugriff auf den Zustand eines Kreis*objektes* = 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();
```

[11] Das ist die offensichtliche Art sich neue Objekte zu schaffen. Eine weitere ist als Literale in Spezialfällen (Strings, Arrays).

## Selbstreferenz: `this`

- spezielle read-only Objektreferenz: Schlüsselwort `this`
- meint dynamisch das momentan-laufende, aufrufende Objekt
- implizites Argument in allen Methodenaufrufen
- Methodenaufruf `o.method(<parameters>)`
  - `o.method()` entspricht oberflächlich `method(o)`
  - Referenz von `o` innerhalb von `method` mit `this`
- Aber: Methodenaufruf ist kein Prozeduraufruf: dynamische Bindung
- innerhalb einer Methode: `this` meist nur implizit: anstelle Beispiels 4 äquivalent möglich:[12]

```
public class Circle3 {
  public double x, y;     // Daten = Koordinaten
  public double r;        // in die Felder

  public double circumference() {
    return 2 * 3.1415926 * this.r;
  };                      //
};                        //
```

[12] Bei Zweideutigkeiten (lokale Variablen oder Namensgleichheit mit einem formalen Parameter) innerhalb einer Methode muß man `this` verwenden, wenn man die nicht-methodenlokale Variable/Methode meinen will.





## Objekte und Klassen: Konstruktoren

- Konstruktor: "Spezialmethode" zur Initialisierung eines Objektes mit dem selben Bezeichner wie die Klasse
- `new` ⇒Aufruf der Konstruktor-Methode.
- Jede Klasse besitzt (mindestens einen) Konstruktor selben Namens, falls keiner angegeben wird ⇒Default-Konstruktor ohne Parameter (siehe vorheriges Beispiel 5)
- mehrere Konstruktoren mit unterschiedliche Parameterlisten möglich: Beispiel für "method overloading".[13]
- kein (expliziter) Rückgabetyp

[13] 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 später
  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 *Klassen*namen: `<classname>.<var_name>`[14]
   - sie entsprechen globale Variablen mit dem Klassennamen als *Diskriminator* (z.B. `Math.PI`)[15]
   - lokale Variablen (innerhalb einer Methode) dürfen nicht statisch sein

[14] Instanzen der betrachteten Klasse dürfen den Klassennamen weglassen.

[15] `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.[16]

```
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
};
```

[16] 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[17]

[17] Grund: `Math` hat viel mit Zahlen zu schaffen, und das sind feste Werte, keine Objekte.





## Initialisierung von Klassenvariablen

- Neben der *Standardsyntax* (Initialisierung bei der Variablendeklaration, z.B. `static int x = 5; ...`) gibt es für komplexere Initialisierungen: static initializers
- Code, automatisch beim Laden der Klasse ausgeführt[18]
- namenlose Methode, kein Rückgabetyp (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)

[18] 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 Systemresourcen, 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 ⇒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 ⇒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.[19] Beachte: dies ist *keine* Mehrfachvererbung

[19] Interfaces kommen erst später.

## Überschreiben von Methoden

- Überschreiben (*overriding*): Kerneigenschaft oo-er Sprachen
- verschiedene Worte: *dynamischer Methodenaufruf*, *dynamic dispatch*, *späte*[20] *Bindung*, *message passing*, meinen alle (so gut wie)[21] das selbe:

  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

[20] d.h. zur Laufzeit

[21] Im Endeffekt laufen alle genannten Dinge auf das selbe hinaus. die Wortwahl kehrt nur verschiedene Aspekte hervor: Binding bezieht sich auf Compilersichtweise. Dispatch und message passing eher die das Verhalten der Implementierung zur Laufzeit.



Sommer 2000 Lehrstuhl für Softwaretechnologie

**Beispiel 10 (Überschreiben)** Hier ein Beispiel, welches Vererbung mit Überschreibung von Feldern und Methoden auf etwas merkwürdige Weise kombiniert.

```
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;
    };
};
```

```
public class Main {
  public static void main (String[] args) {
    B b = new B();
    System.out.println("i = " + b.f());
  };
}
```

Die Ausgabe Beim Ausführen von *main* ist (bis auf die



Sommer 2000 Lehrstuhl für Softwaretechnologie

"newlines"):`i=5; i=2; super.f() = 1; i = 3.`[22] Das erste $i$ ist klar, es ist einfach die in $B$ vorhanden Version von $i$ die die aus $A$ überschreibt (genauer gesagt überschattet (*shadowed*)). Die Zuweisung danach setzt $i$ dann auf 2, wegen `super`, der Wert von $i$ in $A$ ändert sich damit nicht. Das Ergebnis des Aufrufes von $super.f$ ist interessant, es *greift auf die Variable von A zu* und liefert das, wie gesagt, unveränderte $i = 1$. Der Returnwert 3 am Ende sollte damit klar sein.

Wenn wir das Beispiel ändern indem wir die Zeile `int i = 5;` aus $B$ entfernen, erhalten wir folgendes Ergebnis: `i = 1; i = 2; super.f() = 2; i = 4`. Das erste $i$ ist klar: da $B$ das Feld $i$ nicht einführt, erbt es das aus $A$ mit dem Wert 1. Auch die nächste Ausgabe ist klar: $1 + 1 = 2$. Interessant die Ausgabe von $super.f$, diesmal ergibt es den Wert 2, da wir diesmal nur *eine Kopie* des Feldes in $A$ und $B$ haben. Die Rückgabe von 4 ist dann klar.

[22] Das Beispiel während der Vorlesung mit bloß `int i;` war *falsch*, zumindest die erste Ausgabe hatte ich mir offensichtlich falsch notiert. Es verhält sich wie `int i = 0;`, was einleuchtet.



Sommer 2000 Lehrstuhl für Softwaretechnologie

## Statisch vs. Dynamisch

### Beispiel 11

```
class A {
  public int x = 3;
  public void test  () { System.out.println("A"); }
  public void test2 () { System.out.println("x = " +  x); }
};

class B extends A {
  public int x = 4;
  public void test () {  System.out.println("B"); }
  public void test2 () { System.out.println("x = " + x); }
};

public class Main2 {
    public static void main (String[] args) {
        A a = new A();
        A b; b = new B();  // die wichtige Zeile!
        b.test();
        b.test2();
        System.out.println("b.x = " + b.x);
    }
}
```

Dieses Beispiel zeigt klar den Unterschied zwischen statischer und dynamischer Bindung und auch den Unterschied zwischen Deklaration einer Referenzvariablen und dem Instanziierung den Objekts: Felder sind statisch gebunden und Methoden dynamisch.[23]

Die Ausgabe des Programms lautet `B; x = 4; b.x = 3`. Der Aufruf der Methode `test` an $b$ liefert die Methode in $B$, das selbe für den Aufruf von `test2`. Das interessante (oder krumme) ist der Unterschied zu $b.x$ der dadurch kommt, daß $b$ als vom Typ/Klasse $A$ deklariert ist! Schriebe man `B b` anstelle dessen, wäre die Ausgabe von `b.x` ebenfalls 4![24]

[23]Warum die Designer von Java Felder statisch und nicht wie Methoden dynamisch gebunden haben, ist unbekannt. Beachte: wenn von "*statischer Bindung*" der Feldvariablen die Rede ist, sind *nicht* Feldvariablen mit dem `static` Modifier gemeint, sondern die normalen Instanzvariablen. Statisch meint: wenn der Compiler herausfinden muß, welcher Code bzw. welche Daten zu welchem Namen (hier Methoden- bzw. Feldnamen) gehören, dann legt er diese Zuordnung oder Bindung zur Compilezeit fest, sprich: er schaut in der Klassendefinition nach. Natürlich sind als `static` deklarierte Felder und Methoden in diesem Sinne *statisch* gebunden. Methodenaufruf dagegen ist in oo-Sprachen dynamisch.

[24]Später, wenn wir *Casten* kennengelernt haben werden, kann man den Unterschied auch daran sehen: Deklarierte man `B b;` aber verwendete man in der Ausgabezeile `System.out.println("b.x = "+ ((A)b).x);`, also "$b$, gecasted auf $A$", so bekommt man den Wert des Feldes wie er in $A$ deklariert wird, also 3! Umgekehrt, eine Instanz von $A$ —anders als etwas was man von "Typ" $A$ deklariert hat— kann man nicht nach $B$ casten, denn es führt zu einem Laufzeitfehler.





## Kapselung

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

| erlaubter Zugriff | Sichtbarkeit | | | |
|---|---|---|---|---|
| | `public` | `protected` | default | `pr` |
| selbe Klasse | Ja | Ja | Ja | |
| selbes Paket | Ja | Ja | Ja | N |
| Unterklassen (anderes Paket) | Ja | Ja | Nein | N |
| beliebige Klasse | Ja | Nein | Nein | N |

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[25]
- *protected:* Unwichtige 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

[25]Am besten, man vermeidet es ganz indem man *Zugriffsmethoden* schreibt.

## Abstrakte Klassen

- abstrakte 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[26]
- Nutzen:

17. April 2000

[26] Eine Sonderform des *Überschreibens*.



# Lektion IV

# Ausnahmebehandlung

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

**Literatur:** Den entsprechenden Abschnitt aus [Fla99b] oder Abschnitt 14.1 aus [LL97]. Deneben 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 12 (Division durch Null)

```
class Div {
  public static int division (int z, int n) {
    return z/n;                                    // Zeile 3
  };
};

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 aus `java.lang`.

## Ausnahmen

- Ausnahmen sind Objekte, genauer Instanzen von `java.lang.Exception` (bzw. deren Unterklassen)
- sie implementieren das Interface `Throwable`[27]
- 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 <excpt_class1> <var1> {
    <stmnt-list_1>}{
  catch <excpt_class2> <var2> {
    <stmnt-list_>2}{
....
}
 finally
{
  <stmnt_list_n>
}
```

[27] 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 13 (Division durch Null (II))** Der folgende Code zeigt, wie man den Divisionsfehler abfangen — und hier in eher nicht-empfehlenswerter Weise ignorieren — kann:[28]

```
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");
```

[28] 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 „Rückgabetyp"

```
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 14 (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

**Inhalt:** 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 3 aus [Fla99b]. 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.

- Vorwärts-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 Methoden*parameter* 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: Laufzeit-typüberprüfungen
    - ∗ Syntax: `(type) value`

## Beispiel 15 (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[29] und Konstanten (`static final`)
- Default: Methoden des Interfaces sind `public`
- nicht instantiierbar
- $\cong$ Signatur der Instanzen einer Klasse
- Klasse implementiert ein Interface (`implements`)
  - muss typkonforme Implementierung aller Methoden liefern[30]
  - 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

[29] das Schlüsselwort `abstract` muß nicht angegeben werden, ist aber implizit vorhanden.
[30] Bei mehreren Interfaces: alle Methoden. Falls die Interfaces eine Hierarchie bilden, auch noch deren darüberliegenden Interfaces.





## Pakete

- dreistufige Hierarchie: Pakete $\Rightarrow$Klassen $\Rightarrow$Mitglieder[31]
- Konvention: Paketnamen oft mit "." separiert und Paketnamen = Verzeichnispfad
- Vorschlag zur weltweiten Namensvergabe von Klassen:[32]

  $\underbrace{\texttt{de.uni} - \texttt{kiel.informatik.users.ms.classes}}_{\text{Paket, Verzeichnis users/ms/classes}}.\underbrace{\texttt{class}}_{Klasse}.\underbrace{\texttt{meth}}_{Mitglied}$

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

[31] Members = bisher Methoden und Felder.
[32] 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 45)
  - Paket: über Zugriffsrechte (File/Netz)
  - Klassen im Paket: zugreifbar für alle anderen Klassen im Paket, `public`-Klassen auch außerhalb des Paketes
  - Klassenmitglieder: innerhalb des Paketes, es sei denn sie sind `private`
  - Innerhalb einer Klasse: freier Zugriff der Mitglieder untereinander.
  - Klassenmitglieder einer Klasse $A$ 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[33]
- 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

[33] 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.
- ⇒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_Instanz.new innere_Klasse(...)`
- member-Klassen sind schachtelbar
- etwas Abartiges: Top-level Klasse kann eine Memberklasse erweitern[34]

[34] 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 16 (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 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 Enumaration 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 Variablen und Parameter innerhalb des Blockes[35]
- Verwendung oft als sog. Adapterklassen, speziell im Eventmodell von Java 1.1 (*Event Listener* als Implementierung für *listener interfaces*)

[35] technische Bemerkung: `final` deswegen, weil der Compiler *Kopien* für die lokale Klasse anfertigt.





## Beispiel 17 (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 Enumaration 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 instantiierbar
- Kombination von Klassendefinition und Instantiierung in einem Ausdruck (keine Anweisung)
- Syntax (innerhalb eines Ausdruckes), beispielsweise: die anonyme Klasse implementiert ein *Interface*

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

**Beispiel 18 (Anonyme Klasse)** Im folgendes Beispiel implementiert eine anonyme Klasse das *Interface* `FilenameFilter` (mit Methode `accept`) aus dem Paket `java.io`.[36]

```
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]);
  };
};
```

[36] 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 interepret durch die VM |
| | geschachtelt | K/I | `static` innerhalb einer an top-level Klasse |
| Innere Klasse | Member-Klasse | K | nicht-statisches Mitglied, besitzt ßende *Instanz*, mit Zugriff auf glieder, neue Syntax für `new`, su Statische Mitglieder verboten |
| | Lokale Klasse | K | innerhalb eines Blocks, Zugriff a der der umschließenden Klasse lokale Variable, neue `new`-Syntax |
| | Anonyme Klasse | K | namenlose lokale Klasse, *eine* In Konstruktor |

Tabelle 3: Arten von Klassen





8. Mai 2000



# 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.*`. Daneben Abschnitt 3.3 aus [LL97].

Bemerkung: Da wir nun verstärkt in die Klassen der APIs einsteigen, wird es von nun an 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 z.T. notwendig sein, die Klassenbibliothek selbstständig zu Rate zu ziehen. Die Klassenbibliothek ist online verfügbar.

## Verschiedenes

- I/O nicht im Java-Kern, sondern mittels geeigneter Klassen. relevantes Paket: `java.io`
- einfachste Ausgabe auf `stdout`: `System.out.println(....)`[37]
- Quelle für Input oder Output: Strom (*stream*)
- drei vordefinierte Ströme (byte-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 (wg. Effizienz)
- Leeren des Ausgabepuffers: *flushing*, explizit durch `flush`, meist durch `println`, nicht aber durch `print`.

[37] 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`[38]

| | Input | Output |
|---|---|---|
| Byte | `InputStream` | `OutputStream` |
| Character | `Reader` | `Writer` |

Tabelle 4: Übersicht über Stromklassen

[38] `PrintStream` wird nicht mehr so gern gesehen in 1.1. Lieber `PrintWriter`. Allerdings wird man `PrintStream` nicht so schnell los ...





## Ein/Ausgabe: wichtige Klassen

- `InputStreamReader`: Konversion von byte nach character: Übersetzung nach Unicode
- `BufferedReader`: Liest einen Character-Puffer[39]
- `InputStream`: Oberklasse aller byte Input-Streams
- `FileInputStream`: Lesen von Bytes oder Arrays von Bytes von einem File (= Instanz von `File`).[40]

```
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

[39] Pufferung wg. Effizienz
[40] −1 als EOF-Rückgabe





**Beispiel 19 (Buffered reader)** Das Beispiel zeigt, wie man *Text vom stdin* lesen kann. Um von `System.in` lesen zu können, muß man die Eingabe "puffern" [**Bemerkung:** *`System.in` ist ein `InputStream` (eine abstrakte Klasse), also ein byte-stream. Die Klasse `InputStreamReader`* konvertiert *den byte-input-Strom in einen Zeicheninputstrom (internationalisierung). `BufferedReader` schließlich macht die zweite Konvertierung: von Zeichenstrom in einen gepufferten (d.h. effizienteren) Zeichenstrom. Gepuffert heißt beispielsweise auch: man kann* Zeilen *lesen (`readline()`)*

*Für die* Ausgabe.]

```
import java.io.*;

public class Echo {
    /**
     *  Einfachstes Beispiel fuer Lesen vom stdin.
     **/

  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` ist die wichtigste nicht-Stromklasse
  - Repräsentiert 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. (Zu 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 7 aus [Fla99b], Kapitel 6 aus dem alten [Fla97b], Kapitel 4 aus [Fla97a] oder Kapitel 7 aus [LL97].



## Allgemeines

- Applets: Unterklassen von `java.Applet`[41]
- Applets stehen unter der Kontrolle des *Appletviewers/Browsers*
  - ⇒ keine `main()`-Methode notwendig
  - ⇒ keine Command-line-Parameterübergabe, dafür `<PARAM>`-tags
  - ⇒ 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
  ```

  viele (auch) für Applets wichtige Methoden finden sich in vor allem `Component`
- Sonstiges:
  - Sicherheitseinschränkungen für Applets
  - digitale *Signaturen*

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

## Wichtige Methoden bei Applets

| Name | Ausführung bei | typische Verwendung |
|---|---|---|
| init() | Laden des Applets | Initialisierung, Parameterübergabe. F 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 => Ap | | |
| repaint(_) | | (Component), ruft update auf, we schirm löscht und paint aufruft |
| paint(Grapics) | | Malen (Container) |
| print(Graphics) | | Drucken |

Tabelle 5: Wichtige Methoden "an" Applets





## Applets-Lebenszyklus

[**Bemerkung:** *Es hat sich Zeugs geändert in 1.2:* **size** *gibt es nicht mehr.*]

### Beispiel 20

```
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, getSize().width -1, getSize().height -1);
      // size deprecated in 1.2
    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 | getImage(URL) | Laden von Bildern |
| String | getParameter(String) | für Parameterübergabe aus HTI |
| URL | getDocumentBase() | Url der HTML-Seite |
| URL | getCodeBase() | 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) [42]
- Darstelle des `Frames: show`
- Aufrufen der Applet-`init`-Methode durch `main`

**Beispiel 21 (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() {
      public void windowClosing(WindowEvent e) {
          System.exit(0);};
    });
```

[42] die graphische Komponente Frame kommt später genauer, es ist eine Art top-level Fenster.





```
  };                              // end of main

  public void init() {           // auch als applet verwenbar
    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[43]
- `Grahpics`: abstr. Basisklasse aller „bemalbaren“ Komponenten
- nicht direkt instantiierbar (Konstruktor ist `protected`)
- neue Graphik:
  - `Component.getGraphics()/Image.getGraphics(_)`
  - Kopieren: `Graphics.create(_)`
- Methoden zum Zeichen verschiedener Dinge[44] (Strings, Linien, Ellipsen, Rechtecke ... )
- zwei Modi: Normal oder `XOR` (zweimal `XOR` löscht wieder)

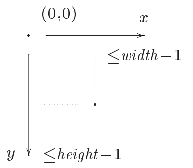



Abbildung 1: Koordinatensystem

[43] die Implementierung der Graphiken ist deviceabhängig, Instanzen von Graphik stellen also ein deviceunabhängige Schnittstelle zur Verfügung. `Graphics2D` ist eine 1.2-Erweiterung von *Graphics*

[44] Bitte selber in der Klasse nachschauen, was es so gibt.





# Graphische Elemente: Fonts, Farben & Cursor

- Font in der Klasse `java.awt.Font`[45]
  - drei Bestimmungsstücke als Konstruktorparameter: Name, Stil, Größe (`String, int, int`)[46]
  - Klasse `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"[47]
  - Konstruktoren mit RGB-Parameter (*rot-grün-blau*)[48]
  - Umrechnung in HSB-Farbmodell möglich
- Cursor in der Klasse `java.awt.Cursor`
  - Verschiedene Standardcursors, insbesondere `DEFAULT_CURSOR`.
  - Übergebbar der Methode `setCursor(Cursor:c)`

[45] Erweitert gegenüber 1.1

[46] Oder über eine "Map" = endliche Abbildung, Assoziationsliste.

[47] `public static final`

[48] Siehe auch `http://www.w3.org/pub/WWW/Graphics/Color/sRGB.html`

# Lektion VIII

# Graphische Benutzerschnittstellen (GUIs)/ AWT

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

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

Die GUI-Libraries haben sich unter allgemeiner Begeisterung und nicht zum ersten zum ersten Mal geändert. Sie haben jetzt die schöne Bezeichnung Java Foundation Classes (JVC) bekommen. Näheres zu Swing vielleicht später.



Weitere Informationen finden sich in Kapitel 8 im alten Nutshell-Buch. Aus Gründen der Geldschneiderei, gibt es nun ein extra-Nutshell buch für die AWT: [Fla99a]. von [Fla99b], Kapitel 6 aus [Fla97a] oder Kapitel 10 aus [LL97]. Events sind eng mit der Programmierung von GUIs verbunden, kommen aber später noch genauer.





## Java Foundation Classes

- Bestehend aus den Paketen:
  - AWT: Modernes GUI-Toolkit, fast aufwärtskompatibel mit JDK 1.1
  - Swing: noch moderneres[49] GUI-Toolkit, erweitert AWT
  - Für *applets*, die von (den meisten) Browsern verstanden werden sollen: besser AWT
  - Java 2D (1.2) Graphic
  - neue API's für Drucken und Datentransfer: (`java.awt.print`, `java.awt.datatransfer`)
  - Applets

[49] „assistive technology“ . . . „pluggable“, „trivially configured“, „pioneering design“ „great step forward“, „PLAF=pluggable-look-and-feel“ . . .





## Übersicht über das Abstract Windowing Toolkit

Wir können vier wichtige Gruppen von Klassen aus dem AWT ausmachen

- Graphik: Klasse `Grapics` zum Pixelmalen (siehe vorangegangenen Abschnitt)
- graph. Komponenten:[50] Klasse `Component` (und `MenuComponent`) und ihre Unterklassen: Interfaceelemente zur Interaktion
- `Component`: wichtiges (abstraktes) GUI-Element
  - Knöpfe
  - Listen
  - Menues
  - Scroll bars
  - Dialoge . . .
- Layout: zur Anordnung von Komponenten innerhalb von Komponenten.
- Events: Ereignisse, zur Interaktion mit der Oberfläche notwendig.

[50] Mit deutschten Worten wie „Komponente“ meine ich hier nur „graphisches Dings“, `Component` sei die konkrete Javaklasse.

## Erstellen einer GUI

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

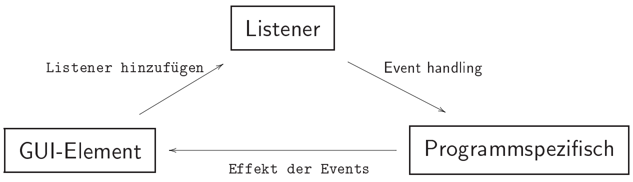



Abbildung 2: Events & GUIs





## 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 butto |
| `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 Menuleiste. `add(Menu m)` fügt neue Menues hinz |
| **Container** | **Nutzen** |
| `Container` | Wurzel der Containerhierarchie |
| `Dialog` | Dialogboxen |
| `Window` | Top-level Fenster ohne sonstigen Schmuck (selten |
| `Frame` | Fenster mit Rand (Unterklasse von `Window`) [**Bem** *heißt auch: größenveränderlich*] |
| `Panel` | Container innerhalb eines anderen, also nicht Generischer Container ohne eigenes Verhalten, schachtelbar, Oberklasse von `Applet`. |
| `ScrollPane` | einzelne Region mit zwei Scollbars, senkrecht und |





## Containers

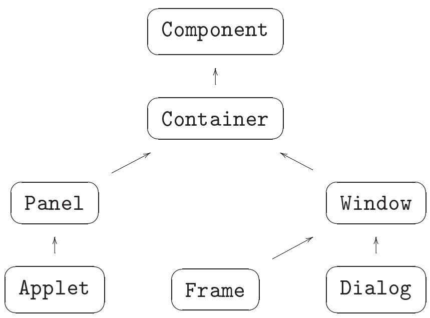



Abbildung 3: Containers

- 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 88).

## Layout Management

- Layout Manager implementieren `LayoutManager` interface[51]
- Wichtige Gruppe von Klassen im `awt`
- Verwendung: Instanzen werden als Argument der Containermethode `setLayout(..)` gegeben.
- Neben dem default-Verhalten: keine Stategie (= `null` = Layout "per Hand") gibt es vordefinierte Manager für verschiedenen Strategien:

| | |
|---|---|
| `FlowLayout` | Fließtextartig |
| `GridLayout` | Spalten- und Zeilenanordnung |
| `BorderLayout` | N/S/O/W und einen in der Mitte |
| `CardLayout` | Nur eine Komponente zu einem Zeitpunkt |
| `GridBagLayout` | komlexere Strategie |

[51] Erweitert zu `LayoutManager2`



# 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 [Fla99b] 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 vielen unterhaltsamen Unterschiede der AWTs in 1.0, 1.1 und 1.2.



## Übersicht

- GUI-Interaktion: ereignisgesteuert ⇒Kern der Guiprogrammierung: Events
- Eventmodell:[52] „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.[53]
  - und nun:*Java 1.2*: Swing als GUI-Teil der *Java Foundation Classes* (JVC)
  - Nächstes Semester: wer weiß[54]
- Eventmodell: drei Mitspieler im Eventprotokoll
  1. Event:
     - Instanz der Klasse `java.util.EventObject`, bzw. deren Unterklasse `java.awt.AWTEvent`[55]
     - Quelle (`getSource`)
     - verschiedene "Sorten" von Events = verschiedene Unterklassen, Event-Klassenhierarchie
     - Definition eigener Eventklassen möglich

[52] 1.1, 1.2 mit Swing
[53] Webbrowser unterstützen oft nur das Ereignismodell von Java-1.0. (Stand 1999)
[54] Wie war das noch mit Java: "write once, run everywhere, everytime"?
[55] in Java 1.0 war es die Klasse `Event`.





     - neue *Namenskonvention*, Zusammenarbeit mit Java-Beans
     - Varianten jeder Sorte als "Feld-*Konstanten*"
  2. Quelle
     - generiert Events
     - benachrichtigt die (passenenden) Listener durch Methodenaufruf, 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[56]

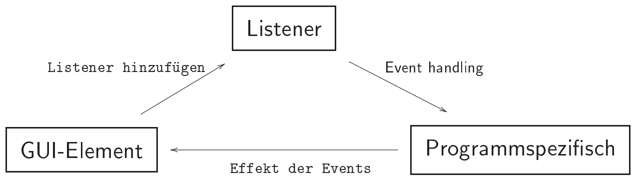



Abbildung 4: Events & GUIs

[56] Ausnahme: der Klasse `MouseEvent` sindl zwei Listener-Schnittstellen zugeordnet: `MouseListener` und `MouseMotionListener`.

# Ereignismodell

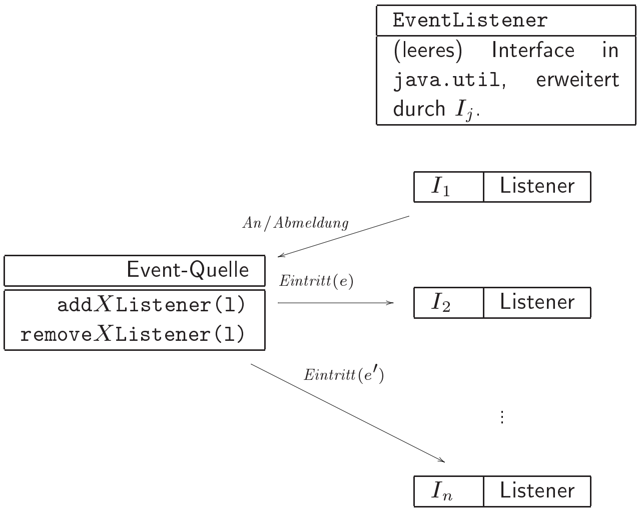



Abbildung 5: Event-Modell (AWT-Events)

[**Bemerkung:** *z.B. das Interface* `java.awt.event.MouseListener` *erweitert* `java.util.EventListener`*. Die Klasse* `java.awt.Component`





*besitzt die Methode* `add.MouseListener` *der der Listener als Argument übergeben wird.*]





# Java 1.0 gegen 1.1

**Beispiel 22 (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 Verwedung innerer Klassen für 1.1

```
import java.awt.*;
import java.awt.event.*;          // Fuer 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");
    };
```





```
  };

  //--------------------Java1.0-----------------
  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 imlementiert werden ⇒ zwei Alternativen:[57]
  1. direkte Implementierung des Interfaces mit `implements`
  2. indirekt mittels Adapter-Klasse
     - Adapter-Klasse: triviale (nicht abstrakte) Implementierung des zugehörigen Interfaces [**Bemerkung:** *Trivial heißt, sozusagen nur stubs.*]
     - Listener: durch Unterklassenbildung und Überschreiben der benötigten Methoden

[57] Die Adapter hier können als eine sehr *einfache* Form der Adapter-Muster [GHJV95] betrachtet werden.





## Eventklassen und Listenerklassen

- Namenskonvention: s. Table 8
- Beispiel 22 auf Folie 106: *Sorte* = `Action`
  - Quelle = `Button` erzeugt Instanzen von `ActionEvent`.1
  - *innere Klasse* `B1` implementiert das Interface `ActionEventListener`.
  - Dies Interface listet auf, auf welche Methoden ein Action-Listener horchen muß ⇒`actionPerformed` enthält den programmspezifischen Kode
  - Registrierung des Listeners `new B1()`an der Quelle: `b2.addActionListener(new B1());`





| | |
|---|---|
| *Sorte*`Event`<br>*Sorte*`Listener`<br>*Sorte*`Adapter`<br>`add`*Sorte*`Listener`<br>`remove`*Sorte*`Listener` | Zu jeder *Sorte* (siehe nächste Folie) gibt es eine Kla<br>Interface der zugehörigen Listener sowie den Adapter.<br>in den Quellen zum An/Abmelder der Listener. Ausı<br>(`MouseListener` und `MouseMotionListener`) |

Tabelle 8: Eventklassen (Übersicht)

| *Sorte* | Methoden | Quelle (nur die höchs chie aufgelistet) |
|---|---|---|
| `Action` | `actionPerformed` | `Button List TextFi` |
| `Adjustment` | `adjustmentValueChanged` | `Scrollbar` |
| `Container` | `componentAdded componentRemoved` | `Container` |
| `Component` | `componentMoved componentHidden componentResized componentShown` | `Component` |
| `Focus` | `focusGained, focusLost` | `Component` |
| `Item` | `itemStateChanged` | `Checkbox, Checkbox Choice, List` |
| `Key` | `keyPressed keyReleased keyTyped` | `Component` |
| `Mouse` | `mouseClicked mouseEntered mouseExited mousePressed mouseReleased`<br>`mouseDragged MouseMoved` | `Component` |
| `Text` | `textValueChanged` | `TextComponent` |
| `Window` | `windowOpened WindowClosing windowActivated windowClosed windowDeactivated windowDeiconifed windowIconified` | `Window` |

Tabelle 9: Eventklassen

15. Mai



# Lektion X

# Threads

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

**Literatur:** [Fla99b] 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. Daneben gibt es natürlich noch [OW97] nur über Threads in Java, aber ich kenne es nicht selbst. Auch in [Lea99] findet sich eine sehr empfehlenswerte Beschreibung von Synchronisation in Java, auch in einem größeren Kontext (insbesondere Abschnitt 2.2 für Locking). 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[58]
- Ein neuer Thread wird — wie üblich — mit `new` erzeugt und durch Aufruf von `start` gestarted.
- Thread endet, wenn das Ende der run-Methode erreicht wird.
- Mit suspend und resume kann die Verarbeitung eines threads angehalten/fortgesetzt werden.

[58] Thread hat verschiedene Konstruktoren.





**Beispiel 23 (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() {                 // "uberschreiben
    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 24 (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");   // Konstuktor 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
- ausführbar
- tot: am Ende der `run`-Methode. Oder auch nach Aufruf von `stop`, wird aber in Java-1.2 missbilligt
- 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:

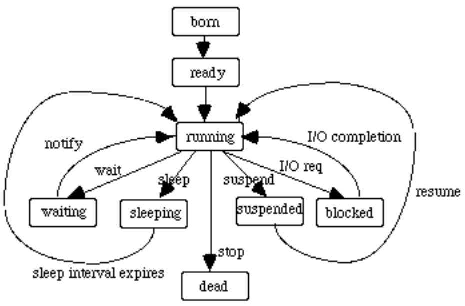



Abbildung 6: Zustände von Threads





## Synchronisation

- Quasi-paralleler Zugriff auf gemeinsame Daten ⇒

  Synchronisation

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

**Beispiel 25 (Leser/Schreiber)** In folgendem Scenario haben die Objekte Schreiber und Leser Zugriff auf dasselbe Speicherobjekt, der eine schreibend, der andere lesend. Das Speicherobjekt soll einfach eine Speicherzelle 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 Ausschlusses in kritischen Abschnitten: Schlüsselwort `synchronized`
- kann
  - Programm-Block
  - Methode

  unter Auschluß 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[59]
- bevor eine synchronisierte Methode ausgeführt werden kann: Lock notwendig ⇒ nur je eine (synchronisierte) Methode kann an der Instanz bzw. der Klasse zu einer Zeit ausgeführt werden

- Aber: Locks erlauben "Reentranten" Kode

[59] nicht nur auf die Methode.





```
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()");
    }
  }
```





**Beispiel 26 (Leser/Schreiber (2))** Da im Leser/Schreiber-Beispiel mittels der beiden Zugriffsmethoden `get` und `put` auf die gemeinsamen Daten zugegriffen werden soll, kann man den gleichzeitigen Zugriff dadurch verbieten, dass man die Methoden als synchronized markiert.

```
public class Speicher {      // Speicher muß 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 gleichzeitigem 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`: Benachrichtigung 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 27 (Leser/Schreiber (3))** In dem Leser/Schreiber-Beispiel kann man diese Methoden verwenden, um die Geschwindigkeit der 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`: zweitweises 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`: wurde (noch) nie implementiert, ein `suspend` ohne resume.[60]

[60] Streng genommen, keine *deprecated* Methode





## Alternative zum Suspendieren

**Beispiel 28 (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` treten 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 Kn"opfe
    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);                      // Kn"opfe 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 ⇒Scheduler
- Javas *Laufzeitsystem*: Scheduling mit fester Priorität[61] (*fixed priority scheduling*)
  - mehrere Threads ausführbar: nimm den *dringlichsten*[62]
  - unterbrechend: es läuft *immer*[63] der ausführbare Thread mit der höchsten Priorität, solange bis er
    - endet[64]
    - 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 (Platformunabhängigkeit)
- Thread erbt die Priorität seines *Erzeugers*

[61] Zwischen `Thread.MAX_PRIORITY` und `Thread.MIN_PRIORITY`. Default: `Thread.NORM_PRIORITY`.

[62] Je kleiner die Zahl, desto höher die Dringlichkeit oder Priorität.

[63] 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.

[64] oder mit `stop` abgebrochen wird, was aber nicht empfohlen wird.





- 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 Netzdienste

**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].



## IP-Adressen & Ports

- IP-Adressen: Adressierungschema 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 (ISO-Schicht 4), d.h., dort stehen Transportschicht-Dienste zur Verfügung
  - 16-bit Port Nummer. $\leq 256$: "wohlbekannte" Ports[65], bis 1024 reserviert.

Anwendungsschicht

port . . . port Transportschicht (ISO-4)

IP-Adresse Network/Internet layer (ISO-3)

[65] z.B. 21 = FTP, 23 = Telnet, etc. Siehe RFC 1700





**Beispiel 29 (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, verwendet für Sockets, . . . ).

```
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);
  };

};
```

[**Bemerkung:** *Bei nicht-existenter Adresse:* *`java.net.UnknownHostException`*]

## Sockets

- Sockets: Endpunkte einer Transportverbindung
- dienen zur Interprozeßkommunikation
- Adressierung der Sockets:

  $$\boxed{\text{Socket} = \text{IP-Adresse} \times \text{Port-Nummer}}$$

  d.h., Socket = Transportschichtadresse, in Java als Instanzen der Klasse `java.net.Socket`
- mehrere Verbindungen über den selben Port/Socket möglich
- zurückgegeben von `ServerSocket.accept`
- zwei Transportprotokolle:[66], 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`

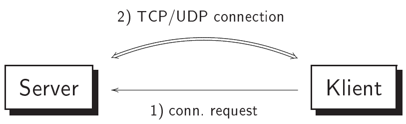
[66] TCP und UDP sind die zwei Internettransportprotokolle *transmission control protocol* und *user data protocol*.





## Client/Servers

- Server stellt Dienste zur Verfügung; im OO-Modell: Methoden
- Klienten fordern Dienste an; im OO-Modell: Methodenaufruf




- Der Aufbau der Verbindung geschieht in zwei Phasen
  1. Klasse `ServerSocket`: Server eröffnet den *Socket* beauf dem er nach Verbindungswünschen von Klienten lauscht. Dieser Port muß allgemein bekannt sein ⇒Methode `accept()`: Server wartet auf Klienten
  2. Klient meldet sich am Port ⇒Aufruf von accept deblockiert[67] und liefert den neuen Socket für die Kommunikation als Ergebnis

[67] 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[68] mit `print` arbeiten will, auch noch `PrintWriter`

[68] Wie z.B. mittels `System.out`





**Beispiel 30 (Server)** Ein einfacher *statischer* Server, der einen Socket aufmacht und zeilenweise Zeichen liest. [**Bemerkung:** *InputStream ist ein Bytestrom, mittels der Klassen* ***InputStreamReader*** *und* ***BufferedReader*** *wird er in einen* gepufferten Zeichenstrom *umgewandelt. Analog für die Writer. Der Port, auf dem gelauscht wird, ist öffenlich, damit ihn die Klienten verwenden können.*]

```
import java.io.*;
import java.net.*;
public class Server {
  public static final int PORT = 2000;     // Portnummer
  public static void main (String[] args) throws IOException {
    ServerSocket s = new ServerSocket(PORT);
    System.out.println("Started: " + s);
    try {
      Socket socket = s.accept(); // Beginn der Bereitschaft
      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);                         // autoflush=on for println

        while (true) {                     // Lesen bis "END"
          String str = in.readLine();      // zeilenweises Lesen
          if (str.equals("END")) break;   // Abbruch
          System.out.println("Echoing: " + str);  // R"uckkanal
          out.println("Echoing: " + str);         // Echo -> stdout
        }; }
      finally {
        System.out.println("closing ...");
        socket.close();
      }}
```

```
    finally {s.close();}};
};
```





**Beispiel 31 (Client)** Hier der zu dem Server aus Beispiel 30 auf Folie 139 passende Klient. [**Bemerkung:** *Den lokalen Host könnte man auch mittels* **`getLocalHost`** *bekommen, wie im Beispiel 29. Der Klient ist im wesentlichen symmetrisch zum Server.*]

```
import java.net.*;
import java.io.*;
public class SingleClient {
  public static void main (String[] args) throws IOException {
    InetAddress addr = InetAddress.getByName(null);  // 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 30 kann nur einen Klienten gleichzeitig bedienen

⇒ Threads
  - der Multi-Server behält den vereinbarten Port im Auge
  - sobald neue Anfrage: Erzeugung ("spawn") eines neuen Server-Threads
  - Socket als Konstruktor-Parameter

**Beispiel 32 (Server)** Hier der entsprechende Code.

```
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 33 (Server)** Hier ein einzelner Serverprozeß. Er bekommt den Socket den er bedienen soll, als Parameter. Ansonsten analog zum Server aus Beispiel 30. [**Bemerkung:** *Die Änderungen zum statischen Klienten: die Initialisierung (Vereinbarung der Sockets, Starten des Threads) ist in den Konstruktor gewandert, die Implementierung des dynamischen Verhalten als Rumpf der* `run`*-Methode.*]

```
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 =                // ganau 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 34 (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 35 (Ein Klient)** Schließlich ein einzelner Klient, der Code ist nicht wesentlich anders als zuvor: [**Bemerkung:** *Jeder Klient ist ein* Thread. `start()` *startet den Thread, d.h. ruft* `run()` *als den Rumpf des Threads auf. Die Änderungen zum statischen Klienten sind analog zu den Änderungen beim Server.*]

```
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 `DatagrammSocket`
- basierend auf UDP
- Klasse `DatagrammPacket`: Instanzen sind, klar, Datagramme, also UDP-SDUs
- zwei Konstruktoren: zum Empfangen und zum Senden

```
DatagramPacket(buf, buf.length)                     // R
DatagramPacket(buf, buf.length, inetAddress, port)  // S
```

- Gesendet werden: Arrays von Bytes (Maximum $<$ 64KByte)
- verbindungsloser Dienst $\Rightarrow$ *keine Analogie* zu `ServerSocket.accept()`
- Codefragmente:
  - Empfangen:

```
DatagramPacket dp = new DatagramPacket(buf, buf.length);
..
dsocket.receive(dp);        //  dsocket = datagram-socket
....dp.getAddress();
....dp.getPort();
....
```

[**Bemerkung:** *Um die Nachricht z.B. Ausdrucken zu können, kann man sie in einen String umwandeln. z.B. indem man den Empfangenen* byte-array *nach Zwischenspeicherung, mittels des Konstruktors*





*`String (buffer, ... )` umwandelt, wobei* `buffer` *z.B. als* `byte[] buffer = new byte[2048]` *vereinbart ist.*]

  - Senden: analog, mit dem zweiten Konstruktor

- Spezialfall: Multicasting

# Multicasting

- Multicasting über Klasse `MulticastSocket`
- 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 Grupenmitglied.
  - `leaveGroup(InetAddress multicastaddress)`: analog
  - reservierte Adressen `224.0.0.1` – `239.255.255.255`[69]
  - Gruppen sind offen: Zum *Senden* an eine Gruppe muß man *kein* Mitglied sein, zum Empfangen schon
- Entscheidend ist
  - Datagramm-Socket als Multicast-Socket beim Empfänger
  - Art des Datagramms: unverändert, aber mit der Multicastadresse

[69] 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 recource allocator*
- eindeutige Adressierung von Netzresourcen (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 | | `getProtocol` |
| Resource-Adresse | Host-IP-Adresse | `getHost` |
| | Datei (inkl. Pfad) | `getFile` |
| | Portnummer (opt.) | `getPort` |
| | Referenz (opt.) | `getRef` |

- URL's als Java-Objekte. Beispiel:[70]

```
String hostname = "www.informatik.uni-kiel.de";
String protocol = "http";
String filename = "/inf/deRoever/SS00/Java/"; // = index.html
URL kursurl     = new URL(protocol,
                          hostname,
                          filename);
```

[70] 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) | `http://www.informatik.uni-kiel.de/~ms` |
| ftp | File transfer | `ftp://ftp.informatik.uni-kiel.de/pub/ki` |
| file | lokale Datei | `/home/ms/.plan` |
| news | Brett/Artikel | `news:cau.ifi.fragen` |
| | | `news:AA4534345@news.informatik.uni-ki` |
| gopher | Gopher | `gopher://gopher.informatik.tu-muenchen.` |
| mailto | Email | `mailto:ms@informatik.uni-kiel.de` |
| telnet | Einloggen | `telnet://snoopy.informatik.uni-kiel.de:` |

Tabelle 10: Verschiedene URL's





## Url-Verbindungen

- Kommunikation mittels Url: öffnen und schließen passender Ströme[71]
- | Methode | Rückgabetyp | Nutzen |
  |---|---|---|
  | `openConnection()` | `UrlConnection` | zum Lesen und Schreiben von und auf die `UrlConnection`. |
  | `openStream()` | `InputStream` | erlaubt das direkte Lesen von einem Url, |
- Verbindungen auf Anwendungsschicht über Urls
  - Instanzen von `UrlConnection`
  - Ergebnis von `openConnection()`, kein Konstruktor[72]
  - 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

[71] 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.

[72] Entspricht ungefähr `accept` bei `ServerSocket` auf Transportschichtebene.



# Lektion XII

# Verteilte Objekte (RMI)

**Inhalt:**

**Literatur:** Eine Einführung findet sich im Nutshellbuch [FFCM99] über Javas sogenannte *Enterprise*-Klassen (Kapitel 3, 8 und 13). Einiges ist anscheinend auch in [Har97] enthalten. Schließlich noch [Lea99].



## Remote Method Invocation: Einleitung

- 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.[73]

[73] *Enterprise*-Programmierung ist ein schicker Name für *verteilte Anwendungsprogrammierung*.

## RMI-Überblick

- im Folgenden: Server: (Java)programm, welches Methoden über's Netz anbietet, Klienten sind die Aufrufer
- 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: Subinterface von `rmi.Remote`
  - Server: implementiert das Interface
  - Nameservice: eigenes Programm/Dämon `rmiregistry`
  - Klient: erkundigt sich beim Nameservice nach den Servern





## RMI-Überblick II

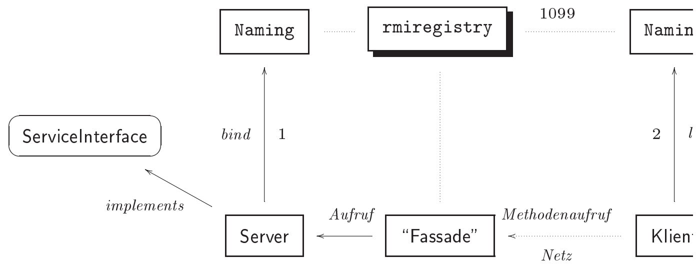



Abbildung 7: Remote method invocation





## Remote Interfaces

- remote interface
  - Erweiterungen des (leeren) Interface `java.rmi.Remote`[74] ⇒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!)[75]
- Implementierung eines RI:
  - Klasse, deren Instanzen die verteilten Methoden zur Verfügung stellen
  - (meist) Erweiterungen von `UnicastRemoteObject`
  - (Sicherheitsregistratur `java.rmi.RMISecurityManager`)

[74] das einzige Interface aus `java.rmi.*`. Der Rest sind der Klassen dient fast nur der Ausnahmebehandlung.
[75] 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[76]
  - `rmiregistry` läuft unabhängig vom Javacompiler/Javaprogrammen, also Obacht beim Starten innerhalb Javas (`LocateRegisty.createRegistry()`)
- Klasse `java.rmi.Naming`: dient zur Kommunikation mit dem Registrationsdämon und zur Namensverwaltung
- Beispiel:

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

  - `snoopy` = Rechner.[77], `ServiceName`: Name (String) unter dem der Service bekannt ist, es muß nicht der Name der Klasse sein
  - `instanz`:
    - ∗ Referenz des *remote objects*
    - ∗ Instanz des UnicastRemoteObject und
    - ∗ Implementierung des RemoteInterface
    - ∗ oft stub
- `rebind()`: dasselbe, aber es ersetzt ggf. einen Service
- Lebenszeit der Dienste:

[76] überwacht i.d.R. Port 1099
[77] man kann auch volle IP-Adressen verwenden

- solange `rmiregistry` läuft
- bis `Naming.unbind()`





## Programmstummel (Stubs) und Skelette

- Client/Server Kommunizieren nicht direkt
- Dem Server muß noch vorgespiegelt werden, daß er lokal aufgerufen wird ⇒
- Programm `rmic`: fügt dem kompilierten Kode (für den `Server`) zwei neue Klassen hinzu:

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

- Stub und Skeleton dienen als Fassade für das RO.
  1. Klient ruft in Wirklichkeit *Stub-objekt* auf
  2. Stub leitet es an Skeleton-Objekt weiter
  3. Skeleton leitet es weiter an das Server-Objekt (und ggf. zurück)





## Verwenden einen RO: Klienten

- Im Gegensatz zum Server: Klientenseite ist einfach
- Lokalisieren des Objektes mittels des `rmiregistry`-Dämons und `Naming.lookup`: [**Bemerkung:** *`Naming.lookup`: gibt ein Objekt des Interfaces `Remote` zurück. Da das leer ist, muß man casten auf das Sub-Interface was man spezifiziert hat. Das Argument ist ein String. Der Rückgabe ist in wirklichkeit der Stub.*]

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

19. Juni 2000



# Lektion XIII

## Datenbankanbindung mit JDBC

**Inhalt:** Einführung · Structured Query Language · Relationale Datenbanen

**Literatur:** Die API zur Datenbankanbindung and SQL-Datenbanken gehört zu den sogenannten *Enterprise*-Klassen, Material findet sich entsprechend in [FFCM99] (Kapitel 2 und Kapitel 18, eine SQL-Superkurzeinführung in Kapitel 8).

Ein fettes Buch zum Thema ist [WFC$^+$99] (da zugehörige online-Material enthält auch ein kleines Tutorial.) Ein Datenbankbeispiel findet sich auch in [Fla97a] (Kapitel 16)

Wir können hier nur sehr oberflächlich auf SQL eingehen, auf der Webseite unseres Kurses sind ein paar weiterführende Informationen aufgelistet.

## Einleitung

- JDBC: Teil der sogenannter Enterprise-Klassen von Java[78]
- dient zum Arbeiten mit relationaler Datenbanken
- Java-Pakete: `java.sql` + Erweiterung `javax.sql`
- Interaktion mittels SQL-Anfragen[79]
- Unabhängig vom Datenbanksystem: Anpassung durch entsprechende Treiber
- Grundschritte jeder Datenbankbenutzung (siehe Beispiel auf Seite 173 (Vorraussetzung: Datenbank läuft bereits)
  1. Laden des Treibers
  2. Erzeugen einer Verbindung
  3. Erzeugen eines Datenbank-Befehls (`Statement`)
  4. Ausführen *einer* SQL-Anfrage
  5. Sammeln und Auswerten der Resultate
  6. Schließen der Resourcen
- Bei mehreren Befehlen: Schritte 3+4+5 müssen wiederholt werden (nicht nur 4 und 5)

[78] Enterprise, in diesem Zusammenhang, ist ein fescher Name für „verteilt".
[79] (*Structured Query Language*), meist SQL-92 bzw. Teilmenge davon.





## Treiber

- Kapselung/Abstraktion der Datenbankimplementierung
- Java-Objekt
- Benuztung: Registrierung beim Manager
  - z.B. mittels `java.sql.DriverManager.registerDriver`
  - am einfachsten/häufigsten: dynamisches Laden + Instanziieren + Registrieren auf einmal mit
    `java.lang.Class.forName (‘‘<Treiber>’’)`
- 4 Arten von Treibern
  1. JDBC-ODBC-Bridge
  2. Native-API- Partly Java
  3. All Java
  4. Native protocol All-Java
- Sammlung von Treibern
- bei uns Postgres-Datenbank mit Typ-3-Treiber





## Verbindung

- Eine Verbindung: Implementierung des *Interface*: `java.sql.Connection`
- Erzeugen: normalerweise mit

  `DiverManager.getConnection()`

  Argument: url, user, password
- Passwortlos: `user = password =`
- am Ende: Schließen der Verbindung mit `close`
- fortgeschrittener Gebrauch: connection pooling/load-balancing





## Befehle

- Statement: zum Ausführen von Datenbankbefehlen
- *Interface* `java.sql.Statement`
- erzeugt mittels `createStatement()` einer `Connection`

```
Statement stmt = con.createStatement();
```

- eigentliches SQL-Statement in Java:

| Code | Anfrage | Beispiel |
|---|---|---|
| `stmt.executeQuery(‘‘SQL’’);` | lesend | SELECT |
| `stmt.executeUpdate(‘‘SQL’’);` | schreibend | UPDATE, DELETE |
| `stmt.execute (‘‘SQL’’)` `stmt.getResultSet ()` | versuchsweise | |

- ein Statementobject = Ein Anfrage
- nach Gebrauch: schließen[80]

[**Bemerkung:** *Daneben gibt es noch:* `getMoreResults()`*, wird aber selten gebraucht, das hängt auch von der Datenbank ab*]

[80] guter Stil, auch wenn das Schließen implizit geschehen sollte, z.B., wenn man eine Verbindung schließt.

## Resultate

- konzeptionell in einer Tabelle (Zeilen/Spalten) aufgebaut entsprechend der gestellten Anfrage
- Objekt vom Interface ResultSet kapselt die Tabelle
- Zugriff:
  1. Zeile
     - sequentiell Zeile für Zeile,
     - Methode next() der Tabelle
     - initial: vor der ersten Zeile
  2. nachgeordnet: Spalte
     - wahlfreier Zugriff
     - (viele) Methoden, je nach Typ
     - insbesondere `getString()`, `getXXX()` über Name der Spalte (d.h. `String`-Argument) oder Index (`int`-Argument von 1 bis $n$).
     - allgemeinster Fall: `getObject()`





## Beispiel (aus [FFCM99])

```java
import java.sql.*;
public class JDBCSample {
  public static void main(java.lang.String[] args) {
    try { // This is where we load the driver
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    }
    catch (ClassNotFoundException e) {
      System.out.println("Unable␣to␣load␣Driver␣Class");
      return;
    }
    try { // access is within a try/catch block.
      Connection con =
        DriverManager.getConnection("jdbc:odbc:companydb", "", "");

      Statement stmt = con.createStatement(); // Create/execute statement
      ResultSet rs = stmt.executeQuery("SELECT␣FIRST_NAME␣FROM␣EMPLOYEES"

      while(rs.next()) {          // Display the SQL results.
        System.out.println(rs.getString("FIRST_NAME"));
      }
      rs.close(); stmt.close(); con.close();  // Freigeben der Rescourcen

    }
    catch (SQLException se) {
      // Inform user of any SQL errors
      System.out.println("SQL␣Exception:␣" + se.getMessage());
      se.printStackTrace(System.out);
    }
  }
}
```





## Bei uns

1. SQL-pakete importieren: `import java.sql.*;`
2. postgres-Treiber einbinden:

```java
try {
    Class.forName("org.postgresql.Driver");
} catch(java.lang.ClassNotFoundException e) {
    System.err.print("ClassNotFoundException: ");
    System.err.println(e.getMessage());
}
```

3. Verbindung zur Datenbank 'JavaUeb' herstellen:

```java
String url = "jdbc:postgresql://sokrates/JavaUeb";
Connection con;
con = DriverManager.getConnection(url, "postgres", "")
```

4. Nach getaner Arbeit die Verbindung schließen: `con.close();`





## SQL + relationale Datenbanken

- SQL
  - Structured Query language
  - standartisierte, deklarative Datenbankanfragesprache für relat. Datenbanken
    - ∗ standartisierte Datentypen (siehe Folie Seite 176)
    - ∗ standartisierte Anfragesyntax
- Wichtigste Datenhaltungsstruktur: Tabelle (*table*), im einfachsten Fall konzeptionell

  array of records

  - Spalten: fester Typ, Name
  - Zeilen
- "relational": individuelle Tabellen können in Relation stehen

⇒ Schema[81]

- Selection: Schema → Tabelle → Spalte: Dot-Syntax:

  `schema_name.table_name.column_name`

- zwei Arten von Befehlen
  1. Manipulation von Schemata
  2. Manipulation von Daten innerhalb von Tabellen

[81] Es gibt noch übergeordnete Strukturen (Cluster und Kataloge) oberhalb von Schemata aber die sind für uns ohne Belang.

## Datentypen: JDBC → Javatypen

| JDBC Type | Java |
|---|---|
| ARRAY | Array |
| BIGINT | long |
| BINARY | byte[] |
| BIT | boolean |
| BLOB | Blob |
| CHAR | String |
| CLOB | Clob |
| DATE | `java.sql.Date` |
| DECIMAL | `java.math.BigDecimal` |
| DISTINCT | mapping of underlying type |
| DOUBLE | double |
| FLOAT | double |
| INTEGER | int |
| JAVA_OBJECT | underlying Java-Class |
| LONGVARBINARY | byte[] |
| LONGVARCHAR | Strirg |
| NULL | ? |
| NUMERIC | `java.math.BigDecimal` |
| OTHER | ? |
| REAL | float |
| REF | Ref |
| SMALLINT | short |
| STRUCT | Struct |
| TIME | `java.sql.Time` |
| TIMESTAMP | `java.sql.Timestamp` |
| TINYINT | byte |
| VARBINARY | byte[] |
| VARCHAR | String |





## Schemamanipulation: Erzeugen

- Wichtigste Schemanmanipulation: Erzeugen einer Tabelle: `CREATE`
- weitere: Verändern/Löschen (`ALTER`, `DROP` . . . )

```
CREATE [ [  GLOBAL | LOCAL ] TEMPORARY]
  TABLE <table_name>
 ( { <column_name> { <data_type> | <domain_name>}
      [<column_size>]
      [<column_constraint>]  ...}
    [DEFAULT <default_value], ..
    [<table_constraint>], ...
    [ ON COMMIT { DELETE | PRESERVE} ROWS ] )
```

- Beispiel

```
CREATE TABLE COFFEES
     (COF_NAME VARCHAR(32),
      SUP_ID INTEGER,
      PRICE FLOAT,
      SALES INTEGER,
     TOTAL INTEGER)
```





- Constraints: Standartbedingungen, per Spalte oder per Tabelle

| Spaltenconstraints (können kombiniert werden) | |
|---|---|
| `PRIMARY KEY` | Hauptspalte (für 0 oder 1 Spalte, keine doppelten Einträge) |
| `NOT NULL` | leerer Eintrag verboten |
| `UNIQUE` | Doppeleinträge verboten |
| **Tabellenconstraints** | |
| `UNIQUE` | keine Doppeleinträge, kann auch für Teilmenge der Spalten definiert sein (UNIQUE(<spalte_a>, ... )) |
| `PRIMARY KEY` | Kombination *mehrerer* Spalten erlaubt! |





## Datenmanipulation

- bei existierender Tabelle: Anfragen und Manipulation einzelner Daten.
- im Wesentlichen:
  - Lesen: `SELECT`
  - Erzeugen: `INSERT`
  - Überschreiben: `UPDATE`

  von Einträgen pro Tabelle

## Lesen

- Lesen mittel SELECT
- Komplexer SQL-Befehle, hier nur beipielhaft

| SQL-code | Erklärung |
|---|---|
| SELECT * FROM BOOKS | alle Spalten |
| SELECT * FROM BOOKS ORDER BY TITLE | Sortieren |
| SELECT TITLE,AUTHOR FROM BOOKS | Teilmenge der Spalten |
| SELECT * FROM BOOKS WHERE PRICE < 10.0 | nach Prädikaten |
| SELECT * FROM BOOKS WHERE PRICE < 10.0 OR EDITION = 1 | log. Verknüpfung |
| SELECT * FROM PERSONS WHERE TITLE LIKE'%Java%' | partieller Stringmatch |
| SELECT * FROM BOOKS WHERE PRICE IN (3.99, 4.99, 5.99) | enthalten in expliz. Aufzählung |
| SELECT * FROM BOOKS WHERE PRICE IN SELECT PRICE FROM PRICES | Schachtelung (Subquery) |

- atomare Prädikate:[82]

  =,<.>,<=,!=,<>, LIKE, IS NULL, IN, BETWEEN

[82] die letzten 4 können mit NOT modifiziert werden





- wichtig Stringmatch mit =, Unterscheidung Groß/Kleinschreibung
- Matching von Teilen mit LIKE
  - _ höchstens ein Zeichen
  - % beliebig viele Zeichen





## Subqueries, Joins, Groups

- Subqueries: geschachteltes SELECT/IN.
- Join Anfragen an mehr als eine Tabelle
  - Restriktion des Outputs gemäß der Relation der Tabellen
  - zwei Arten:
    1. equi-join/Innerer Join
       * symmetrisch
       * gemeinsame Spalte in zwei Tabellen
       * Dot-Syntax oder JOIN-ON-Schlüsselwort
       * Default-join = innerer Join
    2. äußerer Join
       * asymmetrisch
       * zwei Arten: linker und rechter äußerer Join (je nachdem, wo der Primärschlüssel enthalten ist.
       * LEFT JOIN/ RIGHT JOIN





## Join-Beispiele

| SQL-code | Erkärung |
|---|---|
| SELECT * FROM BOOKS WHERE PRICE IN SELECT PRICE FROM PRICES | Schachtelung in einer Tabelle |
| SELECT * FROM CUSTOMERS, ORDERS WHERE ORDERS.CUSTOMER_ID = CUSTOMERS.CUSTOMER_ID | equi-join, Dot-Syntax |
| SELECT * FROM CUSTOMERS INNER JOIN ORDERS ON ORDERS.CUSTOMER_ID = CUSTOMERS.CUSTOMER_ID | equi-join, JOIN-ON-Syntax |
| SELECT * FROM CUSTOMERS LEFT JOIN ORDERS ON CUSTOMERS.CUSTOMER_ID = ORDERS.CUSTOMER_ID | Äußerer Join, Primärschlüssel in Tabelle CUSTOMERS |

# Einfügen

- Einfügen in Tabelle: Schlüsselwort `INSERT`
- Syntax

```
INSERT INTO table_name
[ (column_name),...]
subquery | { VALUES(val1,val2,...) |DEFAULT VALUES
```

- Beispiele:

| SQL-code | Erkärung |
|---|---|
| `INSERT INTO CUSTOMERS VALUES (3, 'Hein Blöd', '0431 454545')` | neue Zeile, genau alle 3 Felder/Spalten |
| `INSERT INTO CUSTOMERS (CUSTOMER_ID, NAME) VALUES (3, 'Hein Blöd')` | selektiv |
| `INSERT INTO JUNKMAIL (NAME, ADDRESS) SELECT NAME, ADDR FROM CUSTOMERS JOIN ADDRESSES` | Kombination mit sub-queries ⇒ mehrere neue Zeilen |





# Überschreiben

- Überschreiben mittels `UPDATE`
- d.h. Modifizierung von 1 oder mehrerer Zeilen
- Syntax

```
UPDATE table_name
SET { column_name = { value | NULL | DEFAULT}, ... }
[ { WHERE predicate}
 | { WHERE CURRENT of cursor_name }]
```

- Beispiele

| SQL-code | Erkärung |
|---|---|
| `UPDATE ADDRESSES SET ADDR = '....', PHONE='...'` | alle Zeilen |
| `UPDATE ADDRESSES SET ADDR = UPPER(STATE)` | Stringmanipulationsfunktion `UPPER` |
| `UPDATE ADDRESSES SET ADDR = '....', PHONE='...' WHERE CUSTOMER_ID = 123` | selektiv |





# Löschen

- Löschen mit `DELETE`
- es gibt kein *undo*
- Syntax

```
DELETE FROM table_name
[ { WHERE predicate }
| { WHERE CURRENT OF cursor_name } ]
```

- Beispiele

| SQL-code | Erkärung |
|---|---|
| `DELETE FROM CUSTOMERS` | alle Kunden löschen |
| `DELETE FROM CUSTOMERS WHERE CUSTOMER_ID > 1000` | selektiv |





# Funktionen

- Ein paar nützliche Hilfsfunktionen:
  - `COUNT(*)`
  - Pro Spalte: `AVG`, `SUM`, `MAX`, `MIN`
  - `CURRENT_DATE`/`CURRENT_TIMESTAMP`: nützlich zum Vergleich mit Datumseinträgen
  - Stringconcatenation: ||
  - Stringkonversion: `UPPER`, `LOWER`
  - `TRIM`: Abschneiden von Zeichen am Anfang/Ende einer Zeichenkette
  - `SUBSTRING`: Teilzeichenkette

4. Juli 2000

# Literatur

## Literatur

[Bud97] Timothy Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley, 2 edition, 1997.

[CW96] Mary Campione and Kathy Walrath. *The Java Tutorial*. The Java series. Addison-Wesley, 1996.

[Eck98] Bruce Eckel. *Thinking in Java*. Prentice Hall, 1998.

[FFCM99] David Flanagan, Jim Farley, William Crawford, and Kris Magnusson. *Java Enterprise in a Nutshell*. O'Reilly, 1 edition, September 1999.

[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.

[Fla99a] David Flanagan. *Java Foundation Classes in a Nutshell*. O'Reilly, 1 edition, September 1999.

[Fla99b] David Flanagan. *Java in a Nutshell*. O'Reilly, 3 edition, November 1999.

[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.

[Lea99] Doug Lea. *Concurrent Programming in Java*, volume 2. Addison-Wesley, 1999.

[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.

[WFC+99] Seth White, Maydene Fisher, Rick Cattell, Graham Hamilton, and Mark Hapner. *JDBC API Tutorial and Reference: Universal Data Access for the Java 2 Platform*. Addison-Wesley, second edition, 1999.