

Java-Programmierkurs

2.-13. Oktober 2000

Ulrich Kortenkamp
Institut für Informatik
FU Berlin

Kursunterlagen ©1997-2000
Jürgen Richter-Gebert, ETH Zürich
Ulrich Kortenkamp, ETH Zürich/FU Berlin

0. Einleitung und Überblick

Themen (I)

Erste Woche:

1. Objekt-orientiertes Programmieren
2. Java als Sprache
3. Das Java-AWT (I) — Grafikoperationen
4. Das Java-AWT (II) — Benutzerschnittstellen
5. Eventhandling in Java
6. Programmierübungen !!!

Themen (II)

Zweite Woche (optional):

1. Java-Beans — Konzepte und Praxis
2. Swing / Java 2
3. Threads (?)
4. Programmierübungen !!!
5. Wunschthemen

“Lernziele”

- Basiswissen über Java
- Einblick in die OO-Philosophie
- Fähigkeit, eigene interaktive Applications und Applets zu programmieren
- Grundlage fürs Selbstlernen

Kursphilosophie

- für Programmieranfänger
- Kleine Lügen erlaubt
- flexibles Tempo
- Fragen erwünscht!
- Programmierübungen sind essentiell
- Lernen am Beispiel
- Nutzt das Web!

Literaturhinweise

- siehe <http://www.inf.fu-berlin.de/lehre/WS99/java>
- Gut: Vieles im Netz
- Eventuell zu empfehlen: Thinking in Java online bei <http://www.bruceeckel.com>

1. Java als Sprache

Was jeder über Java wissen sollte



- Insel im Südpazifik
- Eine der 13000 Inseln von Indonesien
- Einwohnerzahl: 120 Millionen
- Grösse: 50000 km²
- Hauptstadt: Jakarta
- Java hat einen gefährlichen Vulkan namens Merapi
- Exportprodukt: Kaffee (strenger, aromatisch-würziger Geschmack)

Was wir über Java wissen sollten



Java: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language.

Sun Microsystems Inc. über Java

Java is simple and robust

- Wenige Sprachkonstrukte
- Wenige Schlüsselwörter
- *Keine Pointer*
- *Keine Pointer-Arithmetik*
- *Kein GOTO*
- Garbage collection (kein `malloc` und `free`)
- Keine Header-files
- Kein Preprocessor
- Keine Mehrfachvererbung

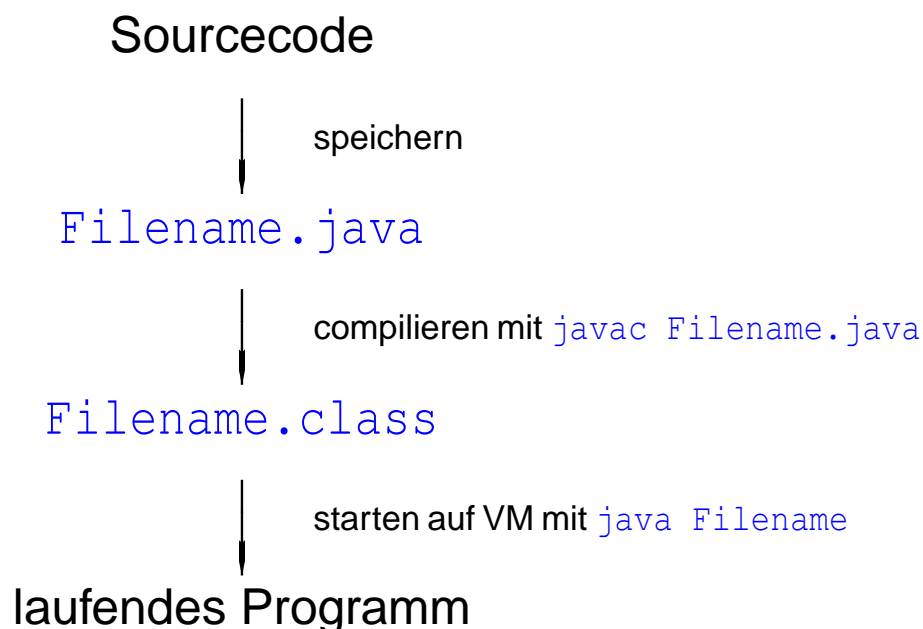
Es wurde darauf geachtet, fehleranfällige Sprachkonstrukte (Pointer, GOTO, Preprocessor, Mehrfachvererbung) gar nicht erst zuzulassen.

Java is interpreted

Java läuft auf einer *virtual machine*. Dies ist ein Programm welches auf der Zielplattform installiert ist und welches *Java-Byte-Code* versteht (die `.class`-files).

Quellcode (die `.java`-files) wird zunächst in Byte-Code übersetzt (compiliert). Dieser läuft dann interpretiert auf der virtuellen Maschine.

Kurzer “Write – Run – Debug” Zyklus:



Java is architecture neutral and portable

- *Virtuelle Maschine* ist auf sehr vielen Plattformen implementiert:
 - Unix (Solaris, Linux, AIX, ...)
 - Windows (95, NT)
 - Netscape
 - Internet Explorer
 - MacOS
 - ...
- Die Programme werden (zur Laufzeit) automatisch an das *Look-and-Feel* der einzelnen Plattformen angepasst.
- *“Write Once, Run Anywhere”*

Java is dynamic and distributed

- Die Interpreterstruktur von Java ermöglicht das *dynamische Nachladen* von `.class`-files.
- Programme erhalten dadurch eine sehr hohe Flexibilität.
- Gut geschriebene (!) Programme können sich somit dynamisch an sich verändernde Bedürfnisse anpassen.
- Nachladen kann *über das Netz* geschehen.

Beispiel:

Ein netzwerkfähiges Zeichenprogramm kann, wenn es ein übers Netz geladenes Objekt zeichnen soll, den Code zum Zeichnen übers Netz dazuladen.

Java is secure

- *Sicherheit* ist erklärtes Designziel der Java-Entwickler.
- Java ermöglicht es, übers Netz geladene Programme in einer abgeschotteten Umgebung laufen zu lassen, in der sie “keinen Schaden” anrichten können.

“Run in a sandbox” Prinzip

- Sicherheitsschutz:
 - keine Pointer
 - kein Zugriff auf Speicheradressen
 - restriktiver Zugriff auf das lokale Filesystem

Java has High-performance

- Java ist eine “Hochsprache” (vergleichbar LISP, SMALLTALK).
- nicht viel langsamer als C (Faktor 3 bis 20).
- *Just-in-time Compiler* ermöglichen zusätzliche Geschwindigkeitssteigerungen.
- Neuere Untersuchungen ergeben eine zu C++ vergleichbare Geschwindigkeit.
- Performance selbst ausprobieren.

“Hello World” Application

Erstellen eines files `HelloWorld.java` mit folgendem Inhalt:

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

Compilieren mit `javac HelloWorld.java`

starten mit `java HelloWorld`

... und man erhält:

Hello World

Applications

application...

... *stand-alone* Programm,

... hat eine `main`-Methode:

```
public static void main(String[] args){...},
```

... hat vollen Zugriff auf das Filesystem,

... virtuelle Maschine muss installiert sein
(z.B. JDK-1.1.7)

Application vs. Applet

applets...

- ... Programm *innerhalb einer WWW-Seite*,
- ... benötigt ein zugehöriges `.html`-file, von dem aus es aufgerufen wird,
- ... ist Unterklasse von `Applet` und somit ein "Panel",
- ... braucht eine `paint` Methode:

```
public void paint(Graphics g){...},
```
- ... wird vom *Web-Browser* oder vom *Appletviewer* aufgerufen,
- ... hat keine vollen Zugriffsrechte.

“Hello World” Applet

Hier ist ein “Hello world” Applet:

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

public class HelloWorldApplet extends Applet {

    private Font font;

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

    public void paint(Graphics g) {

        g.setColor(Color.black);
        g.setFont(font);
        g.drawString("Hello World", 50, 125);
    }
}
```

“Hello World” Applet

Zum Applet gehöriges `.html`-file:

```
<html>
<head>
<title>Hello World Applet</title>
</head>
<body>
I am so proud of my first applet !!

<applet code=HelloWorldApplet.class
width=400
height=200>
Browser does not support Java!!!
</applet>

</body>
</html>
```

Java Sprachumfang

Operatoren:

`++, --, +, -, !, *, /, %, <<, >>, >>>, &, &&, |, ||, ?:`

Vergleiche:

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

Zuweisungen:

`=, +=, -=, *=, /=, %=, <<=, >>=, >>>=, &=, ^=, |=`

Modifier:

`abstract final native private protected
public static synchronized transient volatile`

Types:

`boolean byte char double float
int long short void`

Constants:

`false super this true null`

Control:

`break case catch continue do else for
finally if return switch throw throws try while`

OO-Kern:

`class extends implements imports interface
new package`

Verschiedenes:

`byvalue cast const default future generic
goto inner operator outer rest var`

Vordefinierte Klassen

Java bietet eine sehr grosse Zahl *vordefinierter Klassen* an (z.B. um Graphik, Fenster, IO, Drucken, Maus, Netzwerk, Sicherheit, Mathematik, ... einzubinden).

Unter Rückgriff auf diese Klassen können grosse Programme in kurzer Zeit erstellt werden.

	Java.1.0.2	Java.1.1.7
Pakete	8	21
Klassen	211	503
Methoden	ca. 1000	ca. 5000

Nicht abschrecken lassen

Beispiel: Javas Rectangle (I)

```
public class Rectangle extends Object implements Shape, Serializable {

    // Public Constructors

    public Rectangle();
    public Rectangle(Rectangle r); // 1.1
    public Rectangle(int x, int y, int width, int height);
    public Rectangle(int width, int height);
    public Rectangle(Point p, Dimension d);
    public Rectangle(Point p);
    public Rectangle(Dimension d);

    // Public Instance Variables

    public int height;
    public int width;
    public int x;
    public int y;
```

Beispiel: Java's Rectangle (II)

```
// Public Instance Methods

public void add(int newx, int newy);
public void add(Point pt);
public void add(Rectangle r);
public boolean contains(Point p); // 1.1
public boolean contains(int x, int y); // 1.1
public boolean equals(Object obj); // Overrides Object.equals()
public Rectangle getBounds(); // 1.1
public Point getLocation(); // 1.1
public Dimension getSize(); // 1.1
public void grow(int h, int v);
public int hashCode(); // Overrides Object.hashCode()
public boolean inside(int x, int y); // 1.0
public Rectangle intersection(Rectangle r);
public boolean intersects(Rectangle r);
public boolean isEmpty();
public void move(int x, int y); // 1.0
public void reshape(int x, int y, int width, int height); // 1.0
public void resize(int width, int height); // 1.0
public void setBounds(Rectangle r); // 1.1
public void setBounds(int x, int y, int width, int height); // 1.1
public void setLocation(Point p); // 1.1
public void setLocation(int x, int y); // 1.1
public void setSize(Dimension d); // 1.1
public void setSize(int width, int height); // 1.1
public String toString(); // Overrides Object.toString()
public void translate(int x, int y);
public Rectangle union(Rectangle r);
}
```

Spezielle Klassen

Das OO-Design von Java ist (relativ) streng. Hier ist eine Liste einiger (vielleicht unerwarteter) Klassen:

Object: Diese Klasse ist die *Wurzel* der gesamten Objekthierarchie. Jede Klasse ist Unterklasse von `Object`.

Class: Diese Klasse *repräsentiert selbst wieder eine Klasse*. Die Tatsache, dass Klassen selbst wieder als Objekte repräsentiert sind, ist der Schlüssel zum *dynamischen Laden*.

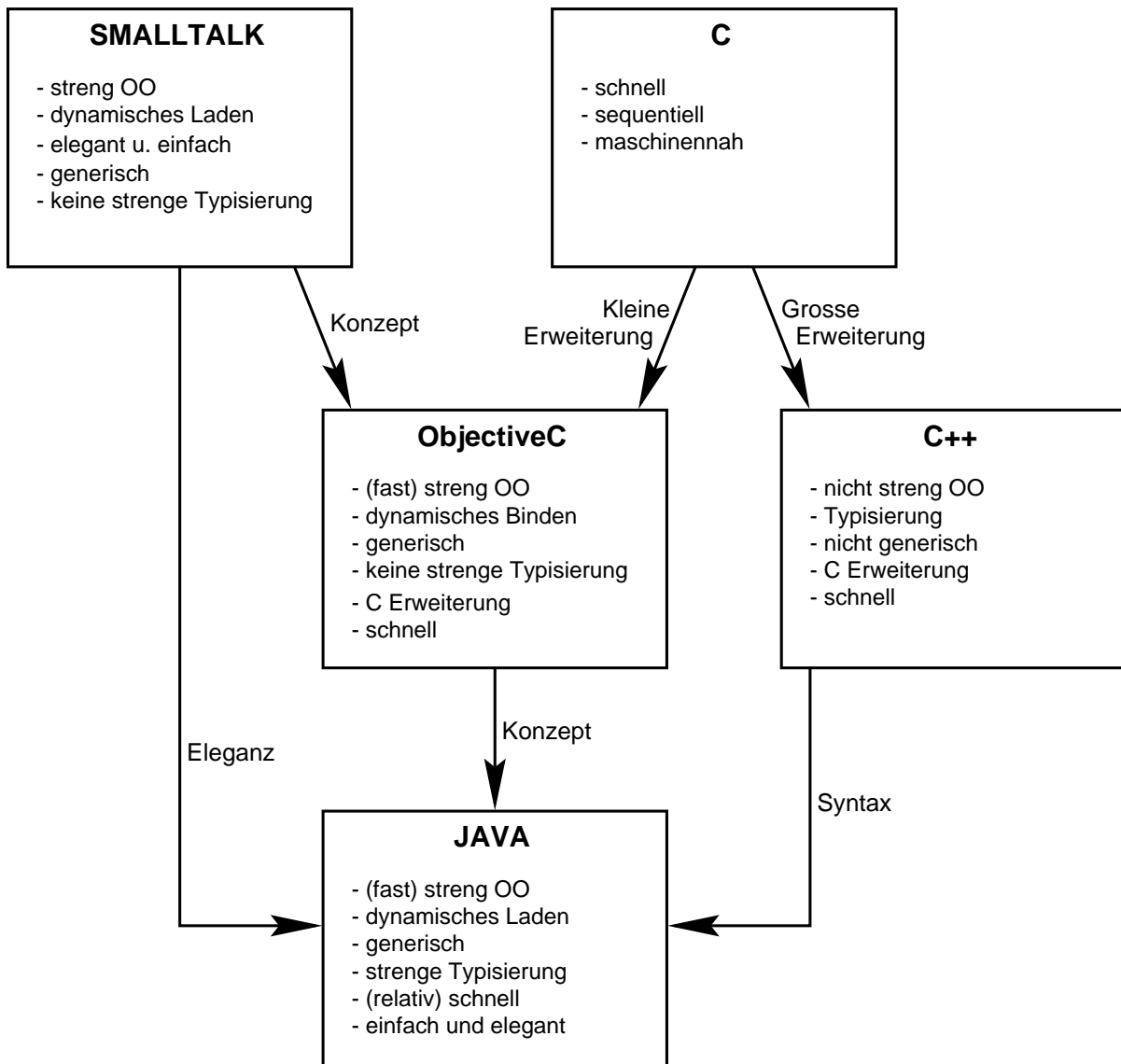
Compiler: Klassencode kann auch dynamisch kompiliert werden. Der Compiler ist hierzu selbst wieder als Objekt repräsentiert.

Error: *Fehlermeldungen* sind ebenfalls als Objekte definiert. Spezielle Fehlermeldungen sind Unterklassen der Klasse `Error`.

Math: *Mathematische Funktionen* (wie `sin`, `cos`, `sqrt`, ...) sind als Klassenmethoden einer Klasse `Math` implementiert.

Thread: Ein von einem Java Programm gestarteter *eigenständiger Unterprozess*. Nützlich (nicht nur) für Animationen.

Stammbaum der Sprachen



2. Grundlagen objektorientierter Programmierung

Objektorientiert vs. sequentiell

Sequentielles Programmieren:

Programm besteht aus *Unterprogrammen*, welche *Daten verändern* und verschiedene *Operationen ausführen*.

Die Daten werden direkt (oder mittels Pointern oder globalen Variablen) von Unterprogramm zu Unterprogramm weitergereicht.

Typische Vertreter: Basic, Pascal, Fortran, Cobol, C, ...

Objektorientiertes Programmieren:

Ein Programm besteht aus einer (aufeinander abgestimmten) Ansammlung von *Objekten*.

Jedes Objekt stellt eine abgeschlossene *Einheit von Daten und Algorithmen* dar.

Typische Vertreter: Smalltalk, ObjectiveC, C++, Oberon, Java ,
...

Warum OO?

Einer von vielen guten Gründen:

Für *fensterorientierte Programmsysteme* ist sequentielles Programmieren zu starr.

Viele Bearbeitungsschleifen müssen geschrieben werden, die alle “Objekte” auf dem Bildschirm (Fenster, Buttons, Zeichenoberflächen, ...) gleichzeitig überwachen und verwalten.

Vorteil von OO:

Objektorientierte Techniken erlauben das *quasiparallele Verwalten* von verschiedenen gleichzeitig existierenden Funktionseinheiten (=Objekten). Die einzelnen Teile kommunizieren miteinander durch Botschaften (messages/methods/events).

Programmieren wird einfacher !

OO-Vokabeln

1. **object**
2. **class**
3. **instance variable**
4. **method**
5. instantiation
6. constructor
7. **inheritance**
8. subclasses
9. **polymorphism**
10. overriding
11. **encapsulation**
12. abstract classes
13. class variable
14. class method

Objekt

Einheit bestehend aus

1. **Daten** (Instanz-Variablen)

Die Beschreibung des konkreten Objektes. (z.B. ein Rechteck mit Grösse, Position, Farbe, ...)

2. **Programmen** (Methoden)

Was kann man alles mit dem Objekt machen? (draw, setSize, move, ...)

Instanz-Variablen + Methoden
= Eigenschaften des Objektes.

Variablen

Um auf Objekte einfach zugreifen zu können, müssen ihnen *Namen* verliehen werden.

Solche Namen müssen zuvor “angemeldet” werden: Mit

```
Color farbe;  
double kontostand;  
String vorname, nachname;
```

werden Variablen namens `farbe`, `kontostand`, `vorname` und `nachname` (durch “,” getrennte Listen zur Abkürzung sind erlaubt) *deklariert*.

Regeln:

- Sinnvolle Namen nehmen
- klein anfangen (`farbe`, nicht `Farbe`)

Zuweisungen

Einer Variablen kann ein Object mit “=” zugewiesen werden:

```
farbe          = Color.red;  
kontostand    = 1000;  
vorname       = "Erwin";
```

Deklaration und Zuweisung können kombiniert werden:

```
String telefon = "030-838 75 159";
```

Scope

Variablen(namen) sind nur innerhalb ihres *Blocks* gültig. Ein Block wird von geschweiften Klammern begrenzt.

Blöcke können geschachtelt werden. In “inneren” Blöcken gelten alle Variablennamen von ausserhalb. Beispiel:

```
{
    Color vordergrundfarbe,
        hintergrundfarbe;
    {
        Color malfarbe = Color.red;
        vordergrundfarbe = malfarbe;
    }
    hintergrundfarbe = Color.red;
}
```

Blöcke treten an diversen Stellen automatisch auf, zum Beispiel bildet die Klassendefinition einen Block.

Hat man keine Chance mehr, auf ein Objekt über einen Namen zuzugreifen, so ist es *weg* — und wird bald aufgeräumt.

Basistypen

Eine Sonderrolle kommt den *Basistypen* in Java zu: Diese sind nicht wirklich Klassen, und die Elemente sind nicht wirklich Objekte.

Name	Zweck	Beispiel
<code>boolean</code>	Ja / Nein	<code>boolean b = true;</code>
<code>char</code>	Buchstaben	<code>char c = 'c';</code>
<code>byte</code>	-128 bis +127	<code>byte i = 56;</code>
<code>short</code>	-2^{15} bis $+2^{15} - 1$	<code>short i = 32767;</code>
<code>int</code>	-2^{31} bis $+2^{31} - 1$	<code>int i = -50;</code>
<code>long</code>	-2^{63} bis $+2^{63} - 1$	<code>long l = 65536*1024;</code>
<code>float</code>	Fließkommazahl	<code>float x = 1 / 3.6;</code>
<code>double</code>	doppelt genau	<code>double y = 3.6 + x;</code>
<code>void</code>	“ohne Typ”	

Kommentare

Java-Programme können *Kommentare* enthalten – diese interessieren den Rechner nicht.

Zwei Möglichkeiten:

- Alles bis zum Ende der Zeile ignorieren mit `//`:

```
boolean hunger = false; // wir sind satt!!
```

- Mehrzeilig ignorieren zwischen `/*` und `*/`:

```
double kontostand=-10000.50; /* wirklich unangenehm,  
aber es ließ sich nicht vermeiden. Nun ja,  
was solls. */
```

Programmcode

Java-Programme, bzw. *Klassencode*:

- Pro Datei eine Klassendefinition
- Immer gleiche Grundstruktur
- In Methoden: Abfolge von *Befehlen*, getrennt durch Semikolon (;)
 - *Deklarationen* `int i;`
 - *Zuweisungen* `i = 10;`
 - *Methodenaufrufe* (gleich!)
 - *Kontrollstrukturen* (später!)

Beispiel: Rechteck

Klassendefinition:

```
class Rectangle extends DiagramObject {
```

Instanz-Variablen:

```
    :  
    private int x,y,w,h;  
    :
```

Methoden:

```
    :  
    public void setPosition(int x,int y) {...}  
    :  
    public void setSize(int w,int h) {...}  
    :  
    public void draw(Graphics g) {...}  
    :
```

```
}
```

Konkreter Klassen-Code

```
import java.awt.*;

class Rectangle extends DiagramObject {

    // DIE INSTANZ-VARIABLEN

    public int x,y;    // Position
    public int w,h;    // Groesse

    // DIE METHODEN

    public void setPosition(int x_pos,int y_pos) {
        x=x_pos;
        y=x_pos;
    }

    public void setSize(int width,int height) {
        w=width;
        h=height;
    }

    public void draw(Graphics g) {
        g.drawLine(x,y,x+w,y);
        g.drawLine(x+w,y,x+w,y+h);
        g.drawLine(x+w,y+h,x,y+h);
        g.drawLine(x,y+h,x,y);
    }
}
```

Klassen / Objekte

Klasse:

“Abstrakte Menge aller Objekte des gleichen Typs”.

Die *Klasse* wird durch den *Programmcodes* festgelegt.

Das Programmstück `Rectangle` beschreibt die Klasse “Rectangle” (Objekte mit Position und Grösse, die man insbesondere zeichnen kann).

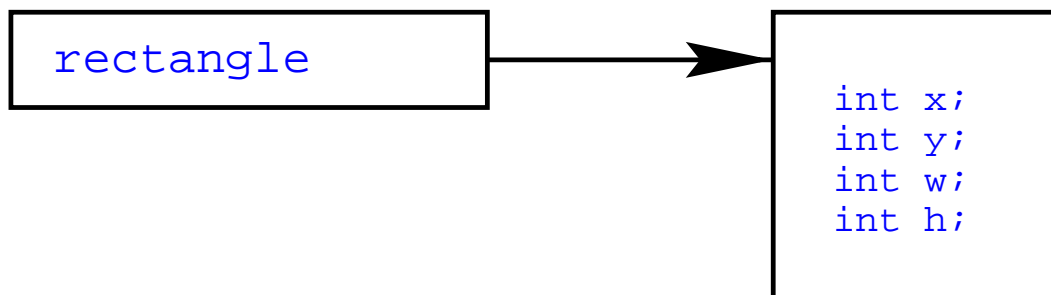
Objekt:

Ein *Objekt* ist eine *konkrete Instanziierung* (Inkarnation) einer Klasse.

(Also z.B. ein konkretes Rechteck mit bestimmten Werten für Farbe, Position und Grösse.)

Intern ist ein Objekt nur ein Zeiger (Pointer) auf einen zugewiesenen Speicherplatz, in dem die Instanz-Variablen stehen.

Beispiel: Objekt Rechteck



Das Objekt `rectangle` ist intern ein Zeiger auf ein Stück Speicher, in dem die Werte der Instanz-Variablen abgelegt sind.

Das Objekt `rectangle` versteht nur “seine eigenen” Methoden (d.h. Methoden, die in seinem Klassencode (und im Code seiner Oberklassen) definiert sind).

Was ist ein Programm ?

**Ein Programm
ist eine Zusammenstellung
von Dateien welche ausschliesslich
Klassencode enthalten.**

**Das gesamte
Verhalten des Programms
wird durch Klassencode festgelegt.**

“Guter Stil”

- *Kurze Methoden* schreiben, die *nur eine* — klar abgegrenzte – Funktionalität realisieren.

(**Faustregel:** keine Methode länger als 20 Zeilen.)

- Klassen schreiben, welche einen speziellen *klar abgegrenzten Objekttyp* isolieren.

(**Faustregel:** Lieber eine Klasse mehr als zwei Methoden mehr.)

- *Pro Klasse eine Quellcode-Datei gleichen Namens anlegen* (z.B. für die `class Rectangle` ein File `Rectangle.java`).

Vererbung

“`class Rectangle extends DiagramObject`” ...

... bedeutet, dass die *Klasse* `Rectangle` alle

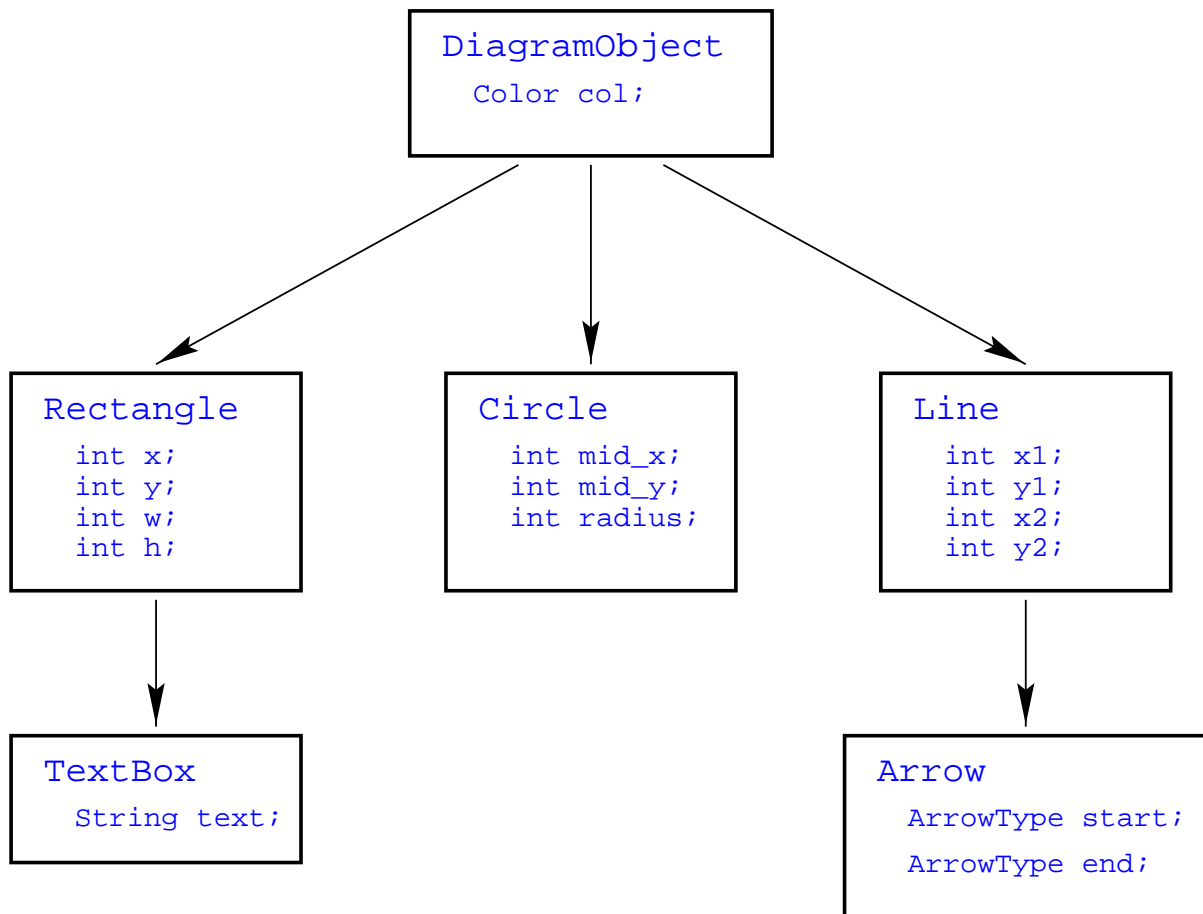
- *Instanz-Variablen* und
- *Methoden*

der *Klasse* `DiagramObject` erbt.

Mittels Vererbung lassen sich Klassenhierarchien erstellen, in denen jede Klasse eine klar abgegrenzte Funktionalität erfüllt.

Das Verhalten einer Klasse ist festgelegt durch den *eigenen Klassencode* und den *Code aller Oberklassen*.

Vererbung



Vererbung der Instanz-Variablen einer kleinen Klassenhierarchie.

Methoden aufrufen

Syntax: Will man von Objekt *obj* die Methode *meth* mit Parametern *param1*, *param2* aufrufen, so schreibt man:

```
obj.meth(param1,param2);
```

Beispiel:

```
// Erzeugen eines Rectangle-Objektes  
Rectangle rect = new Rectangle();  
// Aufruf von Methoden  
rect.setPosition(100,100);  
rect.setSize(50,100);  
rect.draw(g);
```

Zugriff auf Instanz-Variablen

Syntax: Will man auf die Variable *var* eines Objektes *obj* zugreifen,

schreibt man:

obj.var;

Beispiel:

```
// Erzeugen eines Rectangle-Objektes
Rectangle rect = new Rectangle();
// Zugriff auf Instanz-Variablen
rect.x=100;
rect.y=100;
System.out.println("Pos:"+rect.x+", "+rect.y);
```

Erzeugen von Objekten

Konstruktoren ...

... sind *(Klassen-)Methoden*, die Objekte einer Klasse *erzeugen*.

Bei der Erzeugung (Instantiierung) können Parameter übergeben werden und Variablen vorbesetzt werden.

Programmcode für Konstruktoren:

```
Rectangle() {  
    x=0; y=0; w=0; h=0; // direkter Zugriff  
    // auf Instanz-Variablen  
}
```

```
Rectangle(int x_pos, int y_pos,  
int width, int height){  
    x=x_pos; y=y_pos; w=width; h=height;  
}
```

```
Rectangle(Point p,int w,int h){  
    x=p.x;  
    y=p.y;  
    this.w=w; //Vermeiden von Namenskonflikten  
    this.h=h; //durch "this"  
}
```

Aufruf von Konstruktoren

Mit einem *Konstruktor* beginnt der *Lebenszyklus* eines Objektes.

Der Typ eines Objektes muss zunächst deklariert werden, danach kann ein konkretes Objekt erzeugt und zugewiesen werden.

Deklaration und Konstruktoraufruf:

```
Rectangle rect1;                //Typdeklaration
rect1 = new Rectangle(100,100,50,70);
```

Deklaration und Erzeugung können auch in einer Zeile kombiniert werden:

```
Rectangle rect2 = new Rectangle(50,40,10,20);
Rectangle rect3 = new Rectangle(new Point(10,10),
10,20);
```

Lebenszyklus eines Objektes

- Objekt wird mittels Konstruktor *instantiert*,
- mit dem Objekt wird *gearbeitet*,
- wird das Objekt “nicht mehr benötigt”, so wird es vom *garbage collector* (“Müllmann”) zerstört.

garbage collector...

- ... zerstört Objekte, die nicht mehr referenziert werden können,
- ... läuft im Hintergrund,
- ... es wird kein explizites “free” benötigt
- ... am Besten nicht viel darüber nachdenken

Polymorphismus

Derselbe Methodenname kann in *verschiedenen* Objekten benutzt werden.

```
class Line extends DiagramObject {...  
    public void draw(Graphics g) {...}  
...}
```

```
class Rectangle extends DiagramObject {...  
    public void draw(Graphics g) {...}  
...}
```

Der Aufruf erfolgt ganz normal durch *obj.meth()*.

```
Rectangle rect= new Rectangle(10,10,50,50);  
Line line= new Line(0,30,150,50);  
rect.draw(g);  
line.draw(g);  
DiagramObject d = line;
```

Vererbung & Polymorphismus

Die *Kombination* von
Vererbung und Polymorphismus
stellt eines der mächtigsten Programmierwerkzeuge
in objektorientierten Sprachen dar.

Sie bilden die Grundlage
für das Erstellen von *generischem Code*.

Generischer Code ...

... sind Programmstücke, die an die verarbeiteten
Objekte nur minimale Anforderungen stellen
(z.B. "zeichenbar" zu sein).

Denkbar ist z.B. eine Schleife, die an alle auf dem
Bildschirm zu zeichnenden Objekte eine `.draw()`-
Methode schickt.

Überschreiben von Methoden

Unterklassen können die Methoden ihrer Oberklassen *überschreiben* (overloading/overriding).

Beispiel:

```
class DiagramObject extends Object{...
    public void draw(Graphics g){ }
}
```

```
class Rectangle extends DiagramObject{...
    public void draw(Graphics g){
        g.drawLine(x,y,x+w,y);
        g.drawLine(x+w,y,x+w,y+h);
        g.drawLine(x+w,y+h,x,y+h);
        g.drawLine(x,y+h,x,y);
    }
}
```

```
class TextBox extends Rectangle{...
    public void draw(Graphics g){
        super.draw(g);
        g.drawString(text,x+10,y+10);
    }
}
```

Überschreiben von Methoden II

Der Aufruf der Zeichenmethode `.draw` ist identisch für Objekte der Klassen `Rectangle` und `TextBox`.

Beispiel (fortgesetzt):

```
Rectangle rect= new Rectangle(10,10,50,50);
TextBox text = new TextBox(100,100,80,20);
text.setText("Hello World");
...
rect.draw(g); //zeichne Rechteck
text.draw(g); //zeichne Textbox
```

Generischer Code

Beispiel einer generischen `for`-Schleife zum Malen aller Objekte in einem “`Vector`”.

(`java.util.Vector` ist eine vordefinierte Klasse zum Abspeichern einer Liste von Objekten. Die Klasse `Vector` besitzt eine Methode “`.size()`” (Länge der Liste) und eine Methode “`.elementAt(int i)`”.)

Programmfragment:

```
public void drawAll(Vector v, Graphics g){
    for (int i=0; i < v.size(); i++){
        if (v.elementAt(i) instanceof DiagramObject) {
            ((DiagramObject)v.elementAt(i)).draw(g);
        }
    }
}
```

Die Syntax:

(ClassName)Object

teilt dem Compiler mit, dass das *Object* vom Typ *ClassName* ist (*casting*). Ist dies nicht der Fall, entsteht ein run-time-error. In unserem Fall wird mitgeteilt, dass `(DiagramObject)v.elementAt(i)` vom Typ `DiagramObject` ist, und somit insbesondere die Methode `.draw()` versteht.

Kontrollstrukturen

Ein Programm muss nicht immer nur Zeile für Zeile ablaufen. Verschiedene *Kontrollstrukturen* ermöglichen den kontrollierten nicht-linearen Ablauf:

- Entscheidungen
 - `if-else`
 - `switch`-Auswahl
- Schleifen
 - `for`-Schleife
 - `while`-Schleife
 - `do-while`-Schleife
- Abbrüche
- `break`
- `continue`
- `return`

- (`throw`)

if — Entscheidungen

```
if (boolescher Ausdruck) {  
  Block wird nur bei true ausgeführt  
} else {  
  Block wird nur bei false ausgeführt  
}
```

Der `else`-Teil kann auch weggelassen werden:

```
if (boolescher Ausdruck) {  
  Block wird nur bei true ausgeführt  
}
```

Boolesche Ausdrücke

- `boolean`-Variablen
- Vergleiche von Ausdrücken mit `==`
(`true` bei Gleichheit)
- Vergleiche von Ausdrücken mit `!=`
(`false` bei Gleichheit)
- Vergleiche von Zahlausdrücken mit `>`, `<`, `<=`, ...
- Verknüpfung von booleschen Ausdrücken mit
`&&` (und), `||` (oder), `!` (nicht)

Bsp.:

```
if ( ( 3*5 < x) && !( y + x == 4.5) ) {  
    x ist größer als 15 und y ist nicht gleich (4.5-x)  
}
```

for-Schleifen

Eine `for`-Schleife wiederholt einen gegebenen Block, bis eine Abbruchbedingung erfüllt wird.

Syntax:

```
for (init; abbruch?; wiederholung) {  
  zu wiederholender Block  
}
```

Dabei sind *init* und *wiederholung* Befehle, *abbruch?* ist ein boolescher Ausdruck.

init und *abbruch* können auch leer sein, allerdings sind solche Schleifen nicht gut zu verstehen.

for-Schleifen (II)

Die Abarbeitung einer `for`-Schleife erfolgt in der Reihenfolge

init
abbruch?
block
wiederholung
abbruch?
block
wiederholung
abbruch?
...

bis die Abbruch-Bedingung *nicht* mehr erfüllt ist.

Beispiel:

```
for (int i=0; i<10; i=i+1) {  
    System.out.println(i);  
}
```

zählt von 0 bis 9.

Arrays

Wir können viele Variablen des gleichen Typs durch *Arrays* deklarieren. Die einzelnen Elemente des Arrays können über Indizes (*int*-Werte) angesprochen werden.

Beispiel:

```
Color[] farben;
```

vereinbart, dass *farben* ein Array von *Color*-Objekten ist.

Arrays selbst sind Objekte, und man erzeugt ein solches Array zum Beispiel mit

```
farben = new Color[15]
```

Nun können wir 16 (die Nummerierung beginnt immer bei 0!) Farben speichern, zum Beispiel mit:

```
farben[0] = Color.red;  
farben[1] = Color.blue;
```

abstract classes

Man kann eine Methode *meth* einer Klasse *Class* mit dem Modifier *abstract* versehen. Dies bedeutet:

- die Methode *meth* wird in der Klasse *Class* nicht definiert,
- es wird davon ausgegangen, dass entsprechende Unterklassen von *class* das Verhalten von *meth* definieren,
- die Klasse *Class* ist nicht mehr instantiierbar,
- *Class* muss ebenfalls *abstract* deklariert werden,
- nur die Unterklassen von *Class* die *meth* als nicht *abstract* definieren sind instantiierbar.

In unserem Falle wäre es also “guter Stil” die Klassendefinition von *DiagramObject* wie folgt zu ändern:

```
abstract class DiagramObject extends Object {...
    abstract public void draw(Graphics g);
}
```

Interfaces

Natürlich kann es erforderlich sein, dass auch andere Objekte ausser denen der Klasse `DiagramObject` mittels der Methode `.draw()` zeichenbar sein sollen.

Dies kann man durch *interfaces* implementieren.

```
abstract class DiagramObject extends Object
    implements Drawable {
    ...
    abstract public void draw(Graphics g);
}
```

... bedeutet das `DiagramObject` alle Methoden des "Interfaces" `Drawable` implementiert.

Der Code für das *Interface* `Drawable` könnte wie folgt aussehen.

```
public interface Drawable {

    abstract public void draw(Graphics g);
}
```

Interfaces II

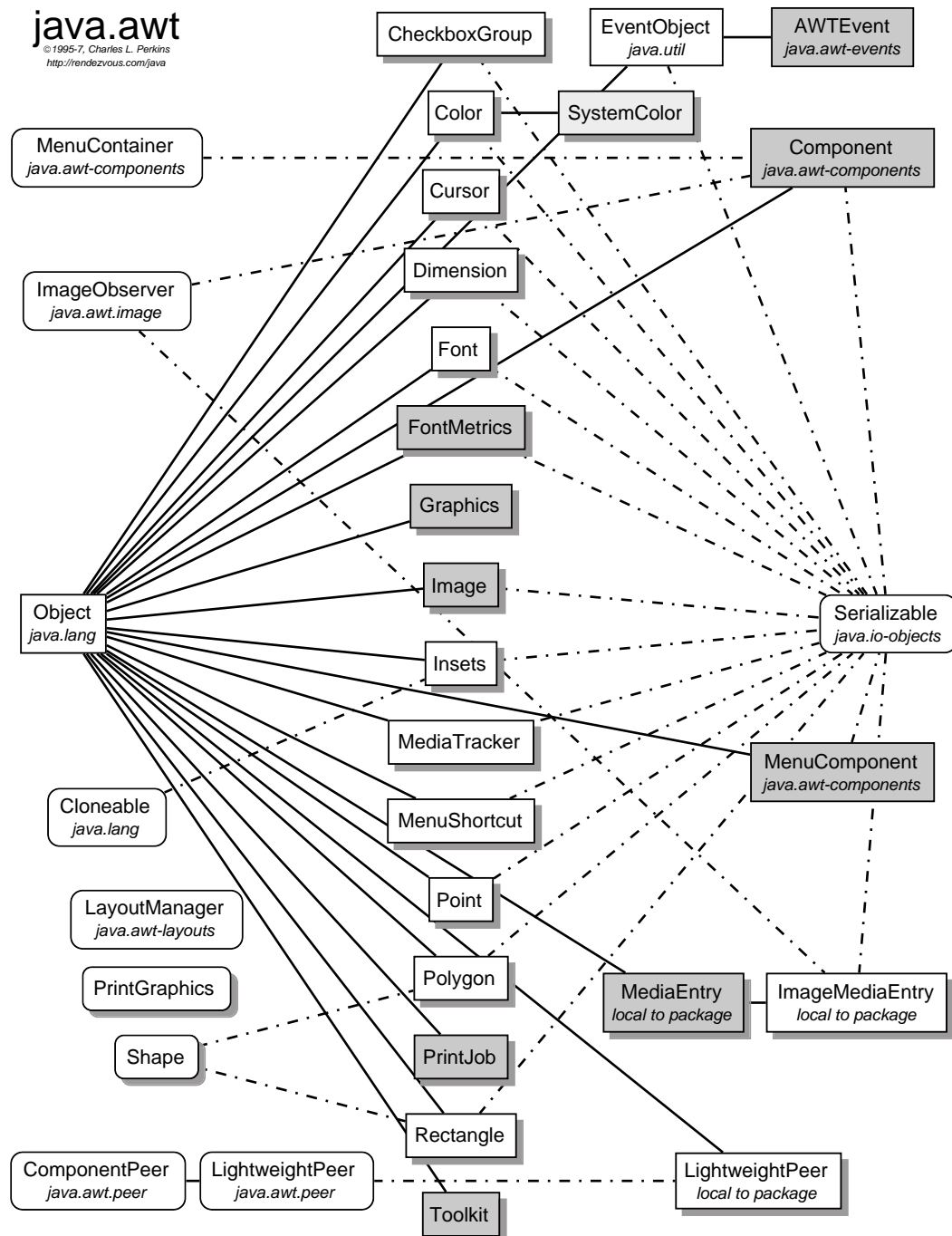
Der Code für unsere `for`-Schleife sollte dann wie folgt geändert werden:

```
public void drawAll(Vector v, Graphics g) {
    for(i=0;i<v.size();i++){
        if (v.elementAt(i) instanceof Drawable) {
            ((Drawable)v.elementAt(i)).draw(g);
        }
    }
}
```

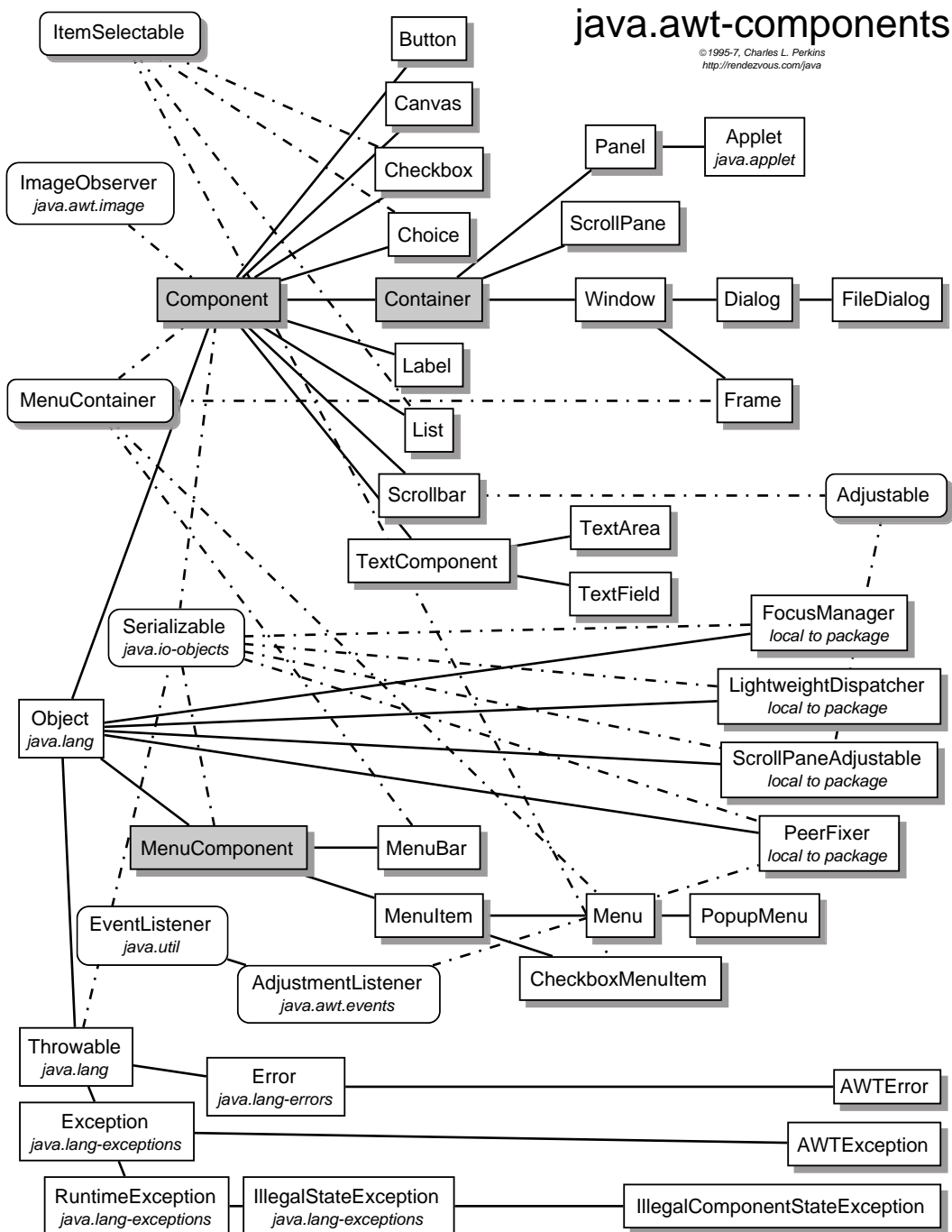
interfaces...

- ... fassen Klassen mit *gleicher Funktionalität* zusammen,
- ... sind *rein deklarativer Natur*, sie legen keinen Code fest,
- ... ersetzen *multiple inheritance* (C++)

AWT: Hierarchie und Interfaces



AWT: Components



Packages

Die Java-Klassen werden in *packages* organisiert. Jede Klasse gehört zu genau einem package, dieses wird mit

```
package paketname;
```

am Anfang der Klassenbeschreibung (in der *.java*-Datei) angegeben. Fehlt diese Paketzuoordnung, so wird diese Klasse automatisch in ein Paket ohne Namen eingeordnet.

Paketnamen sind hierarchisch geordnet, die einzelnen Ebenen werden mit Punkten getrennt. Zum Beispiel werden die Pakete

```
java.awt  
java.awt.event  
java.lang  
java.util
```

mit Java mitgeliefert – wie alle Pakete, die mit *java.* anfangen.

Namenskonventionen

Eine Vereinbarung zur Vermeidung von Namenskonflikten lautet, dass man eigene Klassen in Pakete einsortiert, die mit der umgekehrten *domain* des Programmieres anfangen, hier wäre es also `de.fu-berlin.inf`.

Beispiel:

```
package de.fu-berlin.inf.javakurs;

class Schnuffi {
    ...
}
```

Pakete unterhalb von `java.` sind reserviert – die Virtual Machine sollte keine anderen Klassen in `java.` akzeptieren, ausser denen, die mitgeliefert werden.

Weiterhin gibt es noch `javax.`, dies steht für *Java Extensions*, die nicht direkt zur Java-Plattform gehören, aber standardisierte Erweiterungen sind (Beispiel: Swing, siehe nächste Woche).

Import

Man spricht Klassen bzw. Objekte aus anderen Paketen an, indem man den kompletten Paketnamen voranstellt. So erreicht man die Klasse `Color` aus dem Paket `java.awt` über

```
java.awt.Color
```

Zur Abkürzung kann dem Compiler über `import` mitgeteilt werden, welche Klassen aus anderen Paketen benutzt werden, und, diese können dann ohne Paketbezeichnung erreicht werden:

```
package de.fu-berlin.inf.javakurs;
import java.awt.Color;

public class SuperColor extends Color {
    ...
}
```

Wird bei `import` ein Paketname gefolgt von `.*` angegeben, so kann man jede Klasse des Paketes direkt erreichen. Vorsicht: das kann Verwirrung stiften.

public – protected – private

Java besitzt *Modifier* mit denen die *Sichtbarkeit* von Klassen, Methoden und Instanz-Variablen eingeschränkt werden kann.

`public`:

Die Methode (Instanz-Variable) ist *für jedes andere Objekt* sichtbar (=zugreifbar).

`public` sollten ausschliesslich Methoden sein, durch die das Objekt nach aussen kommuniziert.

`protected`:

Die Methode (Instanz-Variable) ist nur für Objekte des *eigenen packages* und Objekte der *eigenen Unterklassen* sichtbar.

`private`:

Die Methode (Instanz-Variable) ist *nur für Objekte der eigenen Klasse* sichtbar.

`private` bietet Schutz vor Designfehlern. Es ermöglicht das *Ein-kapseln* von Objekten.

default access:

Ohne Modifier ist der Zugriff auf das eigene *package* eingeschränkt.

Rectangle mit private

Hier wurde der direkte Zugriff auf Instanz-Variablen verboten. Zugriff ist nur noch über “get” und “set” Methoden (*Accessoren*) möglich.

```
class Rectangle extends DiagramObject {

    // DIE INSTANZ-VARIABLEN (eingekapselt)

    private int x1,y1    // Position
    private int w,h;     // Groesse
    private int x2,y2;   // Hilfsvariable

    // DIE METHODEN

    public void setPosition(int x_pos,int y_pos) {
        x1=x_pos;
        y1=y_pos;
        x2=x1+w;    //Konsistentes update der
        y2=y1+h;    //Hilfsvariablen x2,y2
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
    .....}

```

Encapsulation

Die *Einkapselung* von Objekten ist eine wichtige Strategie beim Erstellen grosser OO-Programmsysteme.

Nach eingehender Analyse (*Designphase*) legt man genau fest, welche Methoden (und Variablen) einer Klasse nach aussen sichtbar (`public`) sein sollen.

Hierdurch erhält man:

- *wohldefinierte Schnittstellen* zu anderen Objekten,
- *Robustheit* des Programmes, da mögliche Fehlerquellen ausgeschlossen werden,
- *Stabilität*, da Objekte nur in vorgegebener Weise verändert werden können.
- *Flexibilität*, da der innere Aufbau des Objektes geändert werden kann, ohne dass dessen Schnittstelle zu beeinflusst wird.

Anregungen zum Design

- Genaues Überlegen der *Klassenhierarchien*.
- Festlegen notwendiger *interfaces*.
- Überdenken von *Sichtbarkeiten*.
- Überdenken von *Objektabhängigkeiten*:
 - Wer kennt wen ?
 - Wer besitzt wen ?
 - Wer kommuniziert mit wem ?
- Welche *Funktionalität* soll abgedeckt werden ?
- Wie *erweiterbar* ist das Design ?
- Wie *flexibel* ist das Design ?

“Guter Stil”

- Viel `private`, wenig `public`.
- Grosses Gewicht auf die *Designphase* legen.
(**Faustregel:** Lieber drei Tage Design und ein Tag programmieren, als ein Tag Design, vier Tage programmieren, zwei Tage debuggen, ...)
- Möglichst *wenig Code* schreiben.
- Möglichst *allgemeine Ansätze* implementieren.
- Keinen Code duplizieren.
- Keinen Code duplizieren.

