

# Algorithmen und Programmierung II

## *1. Stunde*

- Termine
- Wer bin ich?
- Thema: Objektorientierung und Java
- Thema: Algorithmen

# Algorithmen und Programmierung II

- **Vorlesung**

Dienstag 09:00 - 11:00

Mittwoch 09:00 - 11:00

- **Übungen sind integriert**

- **Praktikum (s. [www.gm.fh-koeln.de/advlabor](http://www.gm.fh-koeln.de/advlabor))**

**Der gesamte Stoff der Vorlesung ist prüfungsrelevant – nicht nur die Themen des Praktikums!**

## Kurze Vorstellung:

Prof. Dr. Erich Eheses

- bis 1979: Studium und Promotion in Hochenergiephysik (Nebenfach Informatik, u.a. Datenerfassung und Simulation)
- bis 1985: Projektverantwortung im Bereich CAD / CAM (u.a. Algorithmen, Sprachen, Compiler, Computergraphik)
- bis 1989: Mitarbeit im Projekt SUPRENUM (u.a. Programmiersprachen, Entwicklungsumgebungen)
- bis 1994: Prof. an der FH Trier (Mitarbeit beim Aufbau des FB Informatik)
- ab 1994: Prof. an der FH Köln / Abt. Gummersbach
- Hobby: Rudern



Prof. Dr. E. Eheses, 2012

# Algorithmen und Programmierung II

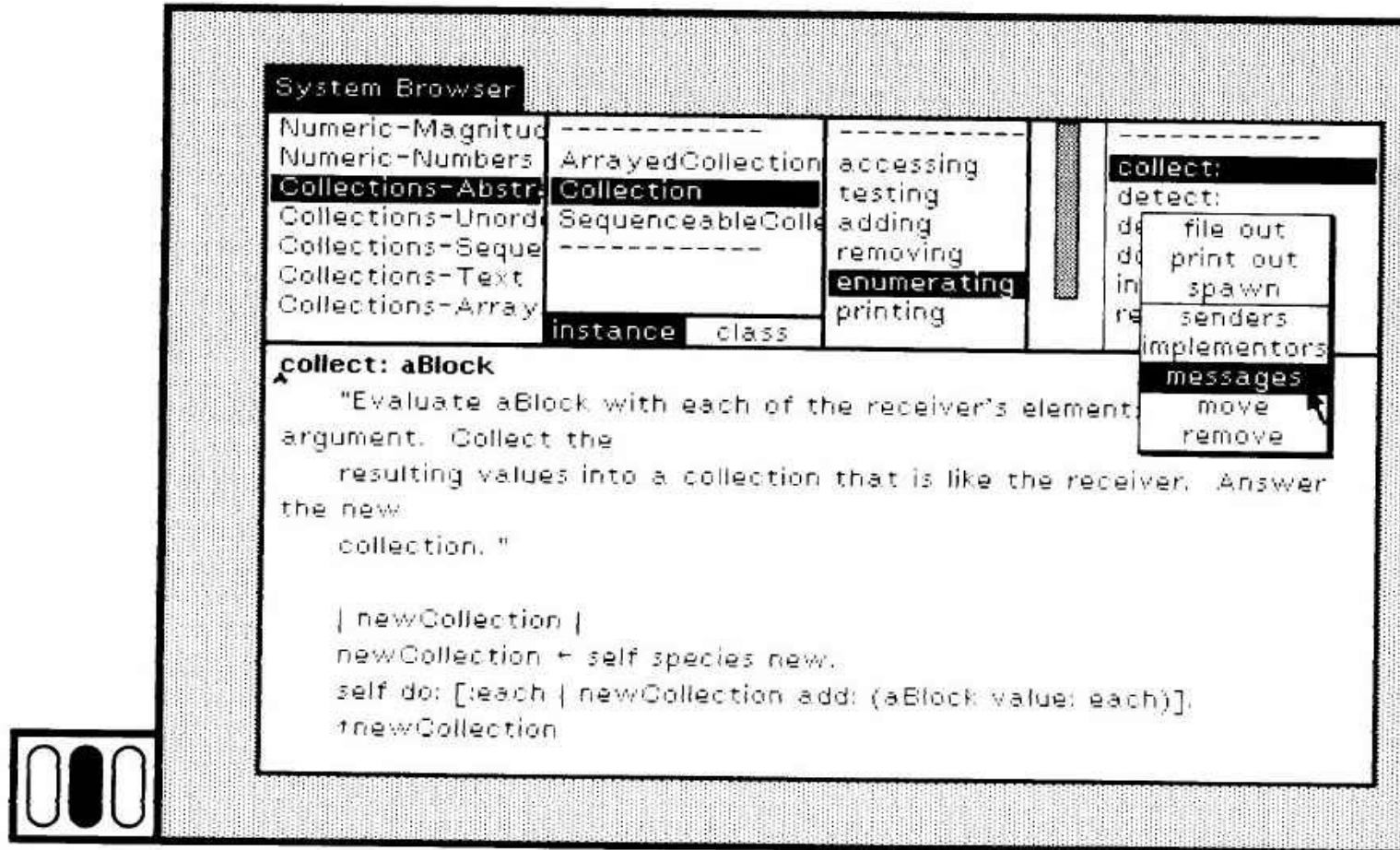
## *Unterlagen*

- **Vorlesungsskript**  
Objektorientierung, Algorithmen, Datenstrukturen
- **Homepage** (Beispiele, Links, Unterlagen)  
<http://www.gm.fh-koeln.de/ehses/ap>
- **David Eck: Programming using Java**  
<http://math.hws.edu/javanotes>
- **Diverse Java Bücher**
- **Java-Dokumentation**  
<http://http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- **Tutorials / Videos**  
z.B.: <http://eclipsetutorial.sourceforge.net/>

**Vorsicht:** Java-Bücher haben ein anderes Lernziel als die Vorlesung:

- Der allergrößte Teil befasst sich Themen die nicht behandelt werden.
- Dafür fehlen wichtige Vorlesungsinhalte!

## Es war vor 30 Jahren ... OOP braucht Tool-Support!



aus Adele Goldberg, *Smalltalk - The Interactive Programming Environment*, 1984

# Algorithmen und Programmierung II

## Werkzeuge

Für den Einstieg sind Kommandozeilenwerkzeuge gut zu brauchen. Sie haben den Vorteil, dass man jeden Schritt einzeln ausführt und dadurch die Zusammenhänge kennen lernt.

Moderne Entwicklungswerkzeuge unterstützen das Arbeiten mit umfangreichen Projekten, wie sie für die Objektorientierung typisch sind.

## Praktikum:

**Eclipse**: [www.eclipse.org](http://www.eclipse.org)

(Es gibt gleichwertige Alternativen: z.B. netbeans)

Eclipse enthält: Editor, inkrementellen Compiler, Klassenbrowser, Testwerkzeug JUnit, Debugger, Refactoring, ...



# Lernumgebungen

Es gibt praktisch für alle Vorkenntnisse Werkzeuge – also gibt es keine Entschuldigung warum man nicht Programmieren lernen kann :-)

**scratch:** <http://scratch.mit.edu>

Visuelle Einführung in die Programmierung (ab 8 Jahre). Man kann sogar ein Gefühl für Objektorientierung bekommen. (basiert auf Smalltalk)

**BYOB:** <http://byob.berkeley.edu>

(byob = build your own blocks) Erweitertes scratch mit Prozeduren und funktionaler Programmierung. Mittels funktionaler Programmierung kann man Kontrollstrukturen bis hin zur Objektorientierung graphisch definieren.

**Alice:** <http://alice.org>

Eine Lernumgebung für die Programmierung von 3D-Szeneri

**BlueJ:** <http://bluej.org>

Eine Lernumgebung für Java

**Greenfoot:** <http://greenfoot.org>

Eine BlueJ-Erweiterung, die den etwas einfacheren Einstieg bietet.

**DrJava:** <http://www.drjava.org>

Eine interaktive Java-Umgebung für Lernzwecke

Und dann gibt es jede Menge Bücher, Tutorials, Video-Lectures ...

# Algorithmen und Programmierung II

## Übersicht

- **Objektorientierte Programmierung**
- **Spracheigenschaften** (Typbegriff, Vererbung, Schnittstellen)
- Java-**Klassenbibliothek**.
- **Programmiermethodik** (Polymorphie, Frameworks, einfache Muster)
- **Algorithmenbegriff**: Korrektheit und Komplexität.
- Dynamische **Datenstrukturen** (Listen und Binärbäume)
- Stack und Queue als Beispiel für **abstrakte Datentypen**.
- **Bäume** und Algorithmen auf (Binär-)Bäumen
- **Algorithmen auf Feldern** (Suchen und Sortieren).
- **Assoziativer Speicher**.

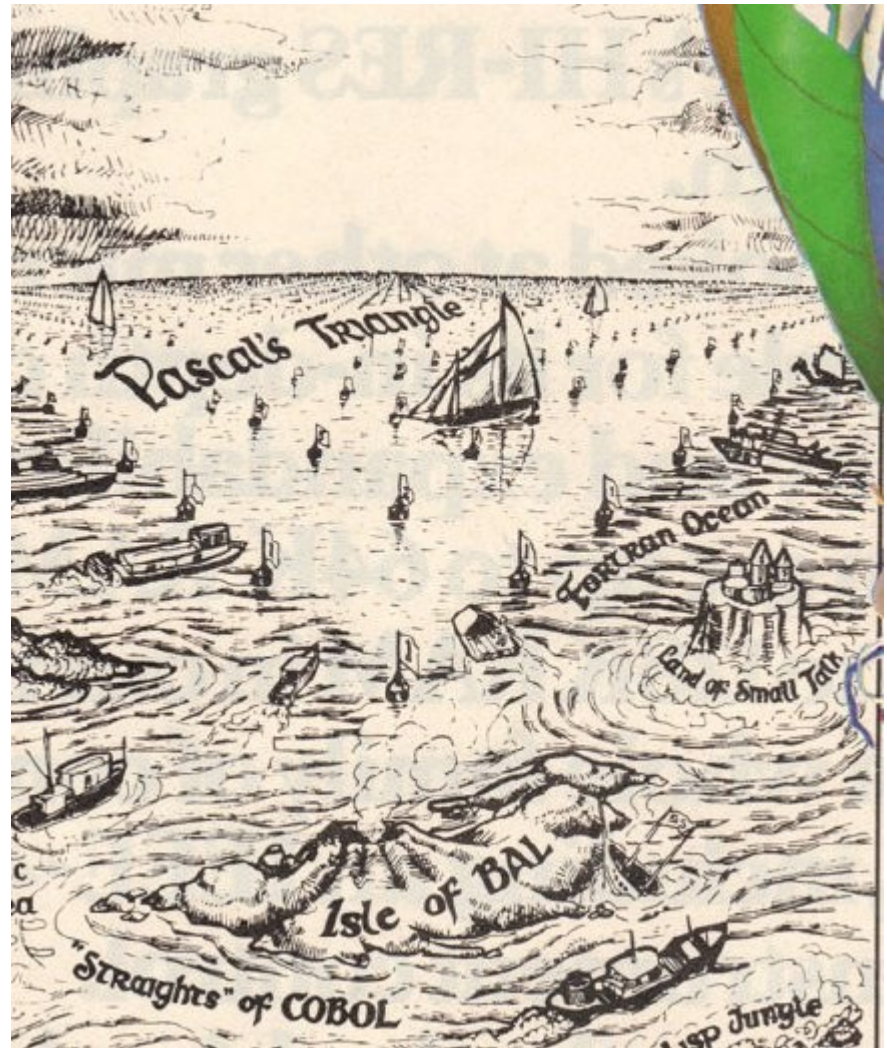
## Der Umstieg auf Objektorientierung

*Traveling upward (in the picture) through heavy seas we come to the pinnacle, a snow white island rising like an ivory tower out of the surrounding shark infested waters. Here we find the fantastic kingdom of Smalltalk, where great and magical things happen. But alas ... the craggy aloofness of the kingdom of Smalltalk keeps it out of the mainstream of things.*

Adele Goldberg, Byte, August 1981

*Many people who have no idea how a computer works find the idea of object-oriented systems quite natural. In contrast, many people who have experience with computers initially think there is something strange about object-oriented systems.*

David Robson, Byte, August 1981



## **Thesen:**

Ein prozedurales Programm ist eine Folge von *Befehlen*.

Ein objektorientiertes Programm besteht aus *Objekten*.

Eine Klasse beschreibt, wie Objekte reagieren.

Wer prozedural denkt, versteht (gute) Klassen nicht.

**Für Objektorientierung muss man umlernen !**

**Frage: Wie wird Wiederverwendbarkeit unterstützt?**

# Programmierparadigmen

## Imperative Programmierung („Gebrauchsanleitung“)

- beschreibt einen *Ablauf*
- prozedurale Programmierung
- *System ist nach Aufgaben strukturiert*

## Deklarative Programmierung („Beschreibung“)

- beschreibt *Sachverhalte*
- funktionale Programmierung, Logikprogrammierung
- kennt z.B. keine Zuweisung und keinen Ablauf
- *System ist nach Sachverhalten strukturiert*

## Objektorientierung („Selbständige Teile = Objekte“)

- beschreibt *Schnittstellen*
- Schnittstellen werden durch *Objekte* realisiert
- Objekte werden durch *Klassen* beschrieben
- OOP enthält immer deklarative Elemente
- (OOP in Java enthält auch imperative Elemente)
- *System ist nach Schnittstellen strukturiert*

*Verständliche Programme müssen deklarativ sein!*

## Prozeduraler Ansatz

```
static double dot(double[] v1, double[] v2) {
    checkVectorLength(v1, v2);
    double r = 0.0;
    for (int i = 0; i < v1.length; i++)
        r += v1[i] * v2[i];
    return r;
}

static void checkVectorLength(double[] v1, double[] v2) {
    if (v1.length != v2.length)
        throw new IllegalArgumentException();
}
```

**Plus:** Einfache Übersetzung. Man kann die Lösung Schritt für Schritt nachvollziehen.

**Minus:** Man **muss** die Lösung Schritt für Schritt nachvollziehen.

**Fazit:** Kleine Programme sind leicht, große Programme nur schwer zu verstehen.

**Wiederverwendbarkeit:** Algorithmen sind für einen konkreten Typ definiert!  
(Ausnahme: ungeprüfte Operationen in C)

## Funktionaler Ansatz

```
def dot(a: Seq[Double], b: Seq[Double]) =  
  (a, b).zipped.map( (x,y) => x*y ).sum  
  
println(dot(List(1.4, -3.2), List(8.2, 2.5)))
```

**Plus:** Die Lösung ist gleich einer Formel:

*Das Skalarprodukt ist die Summe der paarweisen Produkte zweier Vektoren  
Funktionale Programmierung erlaubt die Formulierung abstrakter Ideen*

**Minus:** a) ungewohnt, b) löst (noch) nicht das Problem großer Software

**Fazit:** Ungewohnte Vorgehensweise lässt manches Einfache kompliziert erscheinen!  
Manche komplizierte Dinge werden aber sehr einfach!!

**Wiederverwendbarkeit:** Man kann an **Funktionen höherer Ordnung** Funktionen übergeben.  
Damit erreicht man **Polymorphie!**

**Beispiel:** mittels map kann man beliebige Operationen auf Daten anwenden.

```
(1 to 100).map( x => sqrt(x) )      // Berechnung von Quadratwurzeln
```

## Objektorientierter Ansatz

```
public class Bruch {
    public Bruch addiere(Bruch summand) { ... }

    @Override
    public String toString() { return zaehler + "/" + nenner; }
    ...
}
```

**Plus:** Wiederverwendbare Abstraktionen

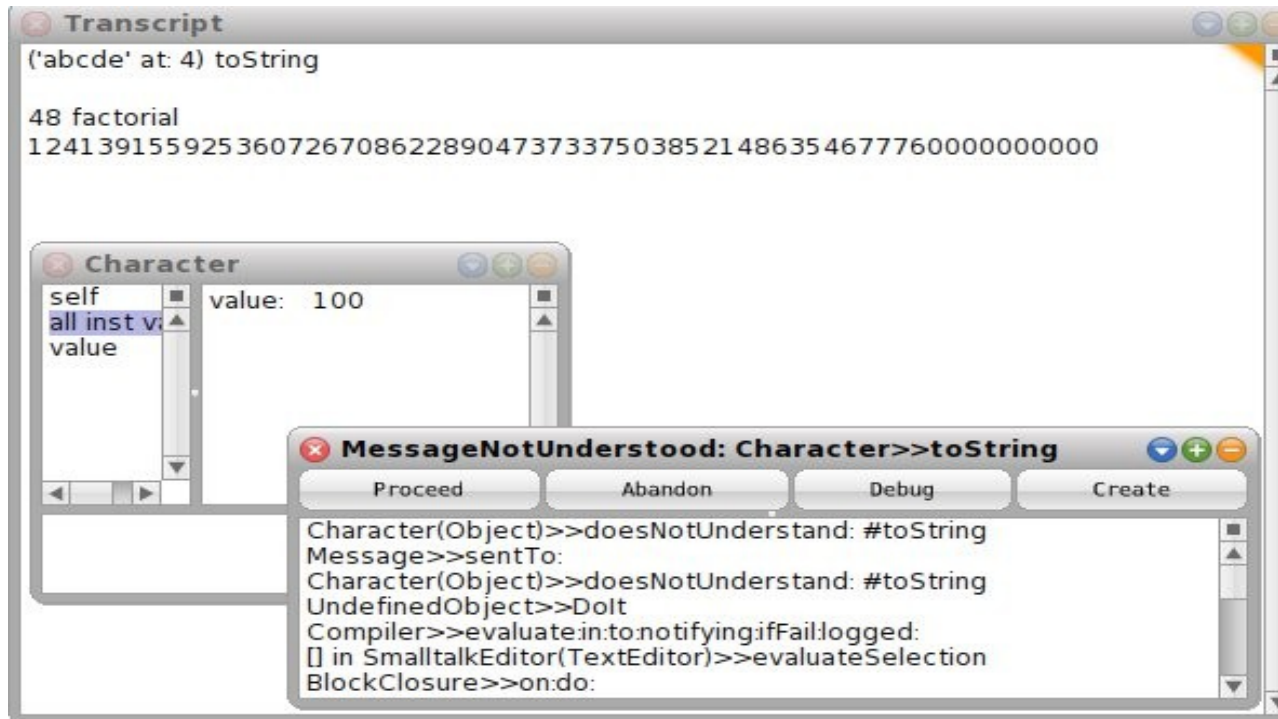
**Minus:** weder Kontrollfluss noch funktionaler Zusammenhang erkennbar

### Fazit:

- Objektorientierung ist ganz anders, als prozedurale Programmierung in C.
- Nicht jedes Java-Programm ist objektorientiert.

**Wiederverwendbarkeit:** Objekte entscheiden nach welcher Methode etwas getan wird  
(Polymorphie)

# Was ist Objektorientierung?



- Objekte verstehen Botschaften (oder auch nicht)
- alles sind Objekte
- Objekte haben eine innere Struktur

## Objektorientierung in Java

Java **verbessert** / **verschlechtert** Objektorientierung durch zusätzliche Eigenschaften: Ähnlichkeit zu anderen Programmiersprachen, möglichst hohe Effizienz, Typprüfung durch den Compiler.

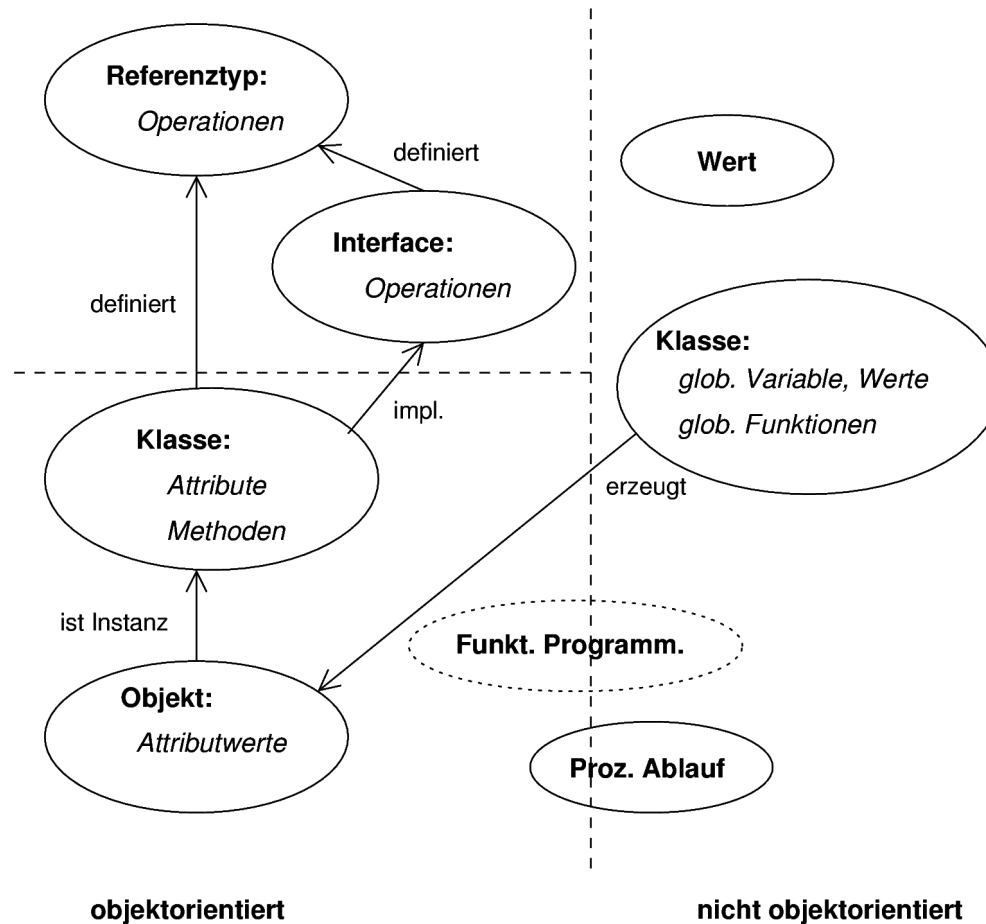
### Besonderheiten:

- **Elementare Datentypen** sind nicht objektorientiert
- **Nicht** „alles ist ein Objekt“ (Closures, Klassen, ...)
- **Klassenfunktionen** sind nicht objektorientiert
- Zugriff auf **Variablen** nicht objektorientiert
- **Typprüfung** durch den Compiler
- **sehr komplexes Typsystem**
- ziemlich „**geschwätzig**“
- „richtige“ **Objektorientierung** von Methodenbindung zur Laufzeit
- sehr hohe **Effizienz** (Optimierung zur Laufzeit)
- hohes Maß an statischer **Überprüfbarkeit**
- sehr gute Unterstützung durch **Werkzeuge**

andere objektorientierte Programmiersprachen machen auch Kompromisse.  
Aber: weniger Kompromisse bei Smalltalk und bei ganz modernen Sprachen.

# Realisierung der Objektorientierung in Java - Themenbereiche

statisches Typsystem



**Java enthält Objektorientierung in besonderer Ausprägung.**

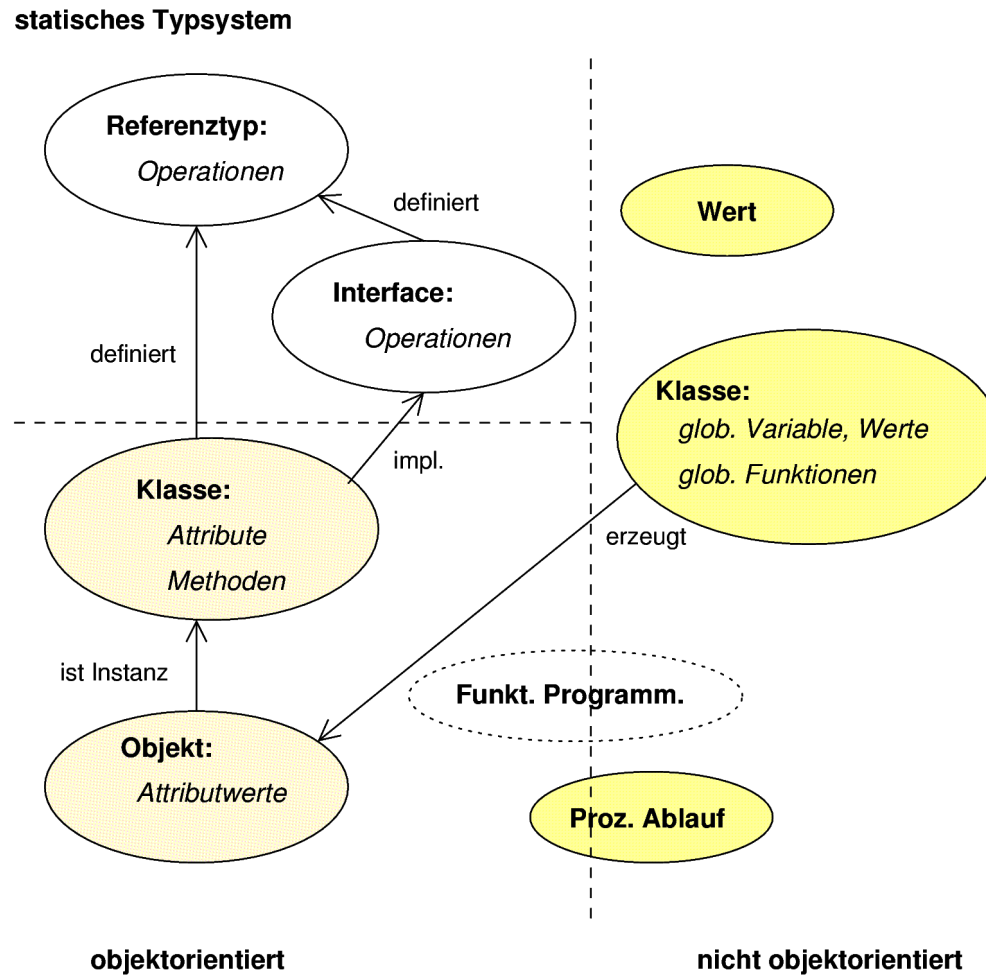
Daraus ergeben sich unterschiedliche Teilbereiche:

- OOP
- Typsystem
- Klasseneigenschaften
- Wertdaten

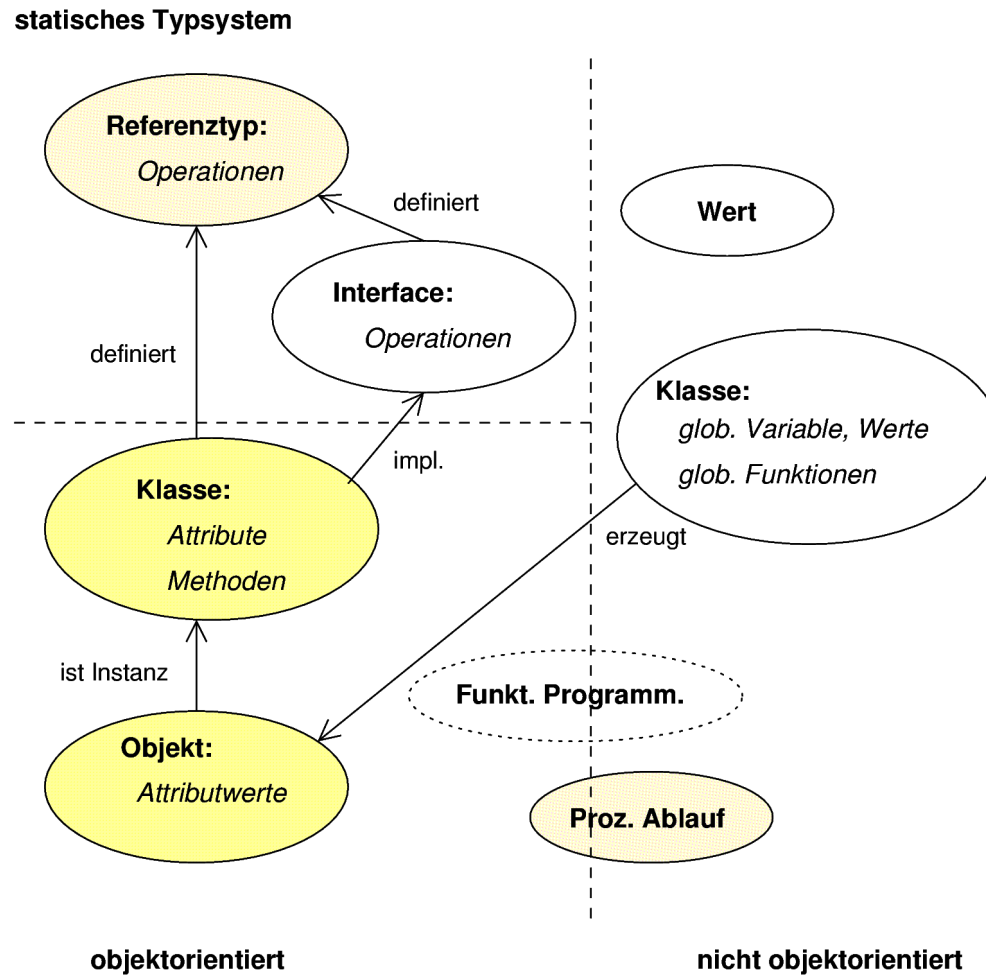
zusätzlich gibt es praktische Aspekte:

- Pakete
- Sichtbarkeit
- Javadoc
- Eclipse
- ...

# Java-Themen des 1. Semesters = prozedurale + klassenbasierte Programmierung



# Eine erste Fokussierung auf Objektorientierung

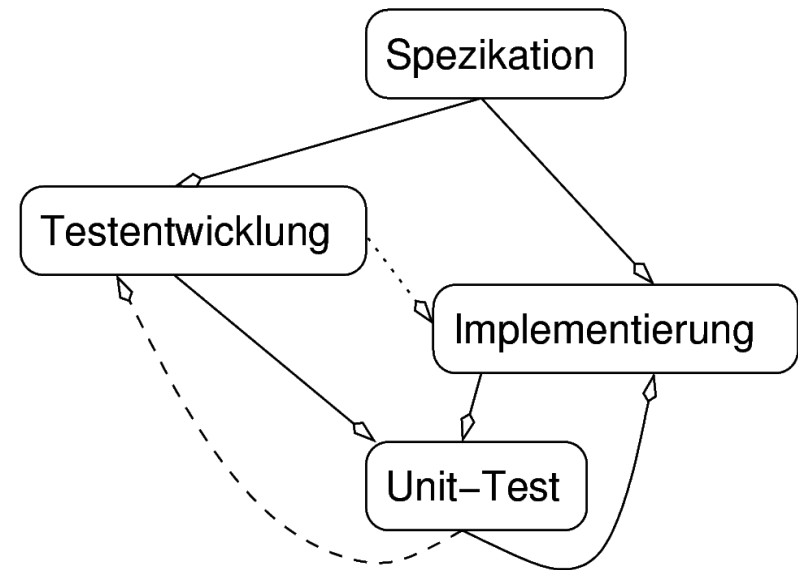


# Objektorientierte Programmierung

- Ein **Objekt** ist ein dynamisches Modul, mit innerem **Zustand** und äußerer **Schnittstelle**.
- Eine **Klasse** beschreibt die gemeinsame **Methoden**-Implementierung von gleichartigen Objekten.
- Objekte werden (in Java) zur **Laufzeit** aus Klassen erzeugt.
- Objektreferenzen werden in **Variablen** gespeichert.
- **Polymorphie** ist die Eigenschaft, dass Objekte verschiedener Klassen gleichartig behandelt werden können (Voraussetzung: Gemeinsamkeiten der Methodenschnittstelle). Polymorphie ist eine Ausprägung von **Abstraktion**.
- Eine Schnittstelle definiert einen **Typ**. Der Typ einer Variablen sagt, welche **Nachrichten/Operationen** die von der Variablen referierten Objekte verstehen.
- Eine Klasse / Objekt **implementiert** einen Typ: nach welcher **Methode** wird etwas gemacht?
- **Typdeklarationen** sind für den **Compiler** da: **frühe Fehlererkennung**.

## Testen von Klassen (Unit-Test)

- man schreibt die komplette Anwendung
- man schreibt für jede Klasse ein `main()`
- man verwendet Eclipse-Scrapbook
- man verwendet eine Testumgebung (JUnit)
- Test setzt eine Implementierung voraus
- Implementierung verlangt die Testdefinition



## **Test-Zuerst (gilt für Unit-Tests)**

Es empfiehlt sich erst den Test und dann die Klasse zu schreiben:

- Das Schreiben des Tests zwingt festzulegen wozu eine Klasse gut ist.
- Durch den Test werden die Aufgaben der Methoden genau festgelegt.
- Ein Testfall sollte zunächst scheitern, damit man die Wirkung des Programms sieht.
- Test-Zuerst führt zu einer großen Testüberdeckung.
- Nach jeder Veränderung einer Klasse werden alle Testmethoden ausgeführt.
- Treten in der Anwendung Fehler auf, die der Test nicht erkannt hat, wird der Test erweitert
- Im AP-Praktikum präzisiert der Test die Aufgabenstellung!
- Das Schreiben eines Tests spart viel Zeit!

Als Werkzeug für automatisierte Klassentests hat sich JUnit durchgesetzt. JUnit steht in Eclipse schon zur Verfügung.

**Unit-Test ist (noch) keine Qualitätskontrolle!**

## Der Aufbau einer JUnit-Testklasse

```
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class BruchTest {
    private Bruch b;

    @Before
    public void initialisierung() {
        b = new Bruch(0);
    }

    @Test
    public void einfacherTest() { // 1. Test
        assertNotNull(b);
        assertEquals(0, b.zaehler());
        assertTrue(b.nenner() != 0, "Nenner darf nie 0 sein.");
    }

    @Test(expected=ArithmeticException.class)
    public void testVonAusnahme() {
        new Bruch(1, 0);
    }
}
```

## Erläuterungen zu JUnit:

Eine **Testklasse** (*test case*) besteht in erster Linie aus **Testmethoden**. Jede Testmethode wird von JUnit separat ausgeführt. Dabei wird zunächst, falls vorhanden, jeweils eine **Initialisierungsmethode** ausgeführt.

Testmethoden sind durch `@Test`, Initialisierungsmethoden durch `@Before` gekennzeichnet. Ausnahmen werden durch `expected=` Tags spezifiziert.

Ein Test besteht aus nötigen Vorbereitungen und aus Überprüfungen von Resultaten mittels **Assert-Funktionen**. Assert-Funktionen stellen Zusicherungen dar, die bei einem korrekten Programm erfüllt sein müssen.:

<code>assertTrue, assertFalse</code>	= die Bedingung muss/darf nicht gelten
<code>assertNull, assertNotNull</code>	= das Resultat muss/darf nicht null sein
<code>assertEquals</code>	= das Resultat muss gleich dem erwarteten Ergebnis sein. (Gleichheit: <code>equals!</code> )
<code>assertSame</code>	= das Resultat muss genau das erwartete Objekt sein.

Assert-Aufrufe haben einen optionalen String-Parameter für eine genaue Fehlermeldung.