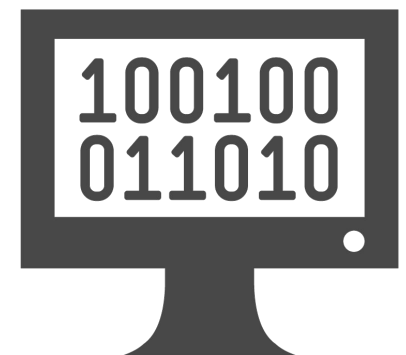


# Fortgeschrittene Programmierungsmethoden



# Struktur

Mihaela Bondor | Mădălina Dicu | Dr. Cătălin Rusu

## Workload (in Stunden):

Vorlesung: 2

Seminar/Labor: 2 + 2

**TEAMS:** Fortgeschrittene Programmierungsmethoden

**Email:** [vasile.rusu@ubbcluj.ro](mailto:vasile.rusu@ubbcluj.ro)

Fragen und Feedback sind immer erwünscht



# Kursinhalt

---

1. Überblick, grundlegende Sprachelemente
2. Objektorientierte Programmierung (OOP)  
***OOP is an island***
3. JAVA und C#
4. Analyse und Design von Softwaresystemen



# Kursinhalt

1. Einführung in Java
2. Klassen und Objekte in Java
3. Collections
4. Input/Output
5. Reflektion
6. Datenbanken
7. Concurrency
8. Funktionale Programmierung
9. Web Programming
10. Metaprogramming
11. XML
12. Einführung in C#
13. LINQ



# Prüfungsform

---

**K**lausur (40%)

**L**ab (30%)

**P**raktische Prüfung (30%)

**Minimale Leistungsstandards:**

**K,P,L  $\geq 5$**

# Einführung in die Programmiersprache Java I



**LEARN JAVA, THEY SAID**



**IT'LL BE FUN, THEY SAID**

# Einstieg in Java

- Download und Installation  
*<https://adoptopenjdk.net/>*
- Entwicklungsumgebung für Java
  - *Eclipse, NetBeans, **IntelliJ***
  - *atom, vim, Sublime*



Version: 2020.2.2  
Build: 202.7319.50  
15 September 2020  
[Release notes](#)

[System requirements](#)  
[Installation instructions](#)  
[Other versions](#)

## Download IntelliJ IDEA

Windows **Mac** Linux

### Ultimate

For web and enterprise development

[Download](#)

Free 30-day trial

### Community

For JVM and Android development

[Download](#)

Free, open-source

	IntelliJ IDEA Ultimate	IntelliJ IDEA Community Edition ⓘ
Java, Kotlin, Groovy, Scala	✓	✓
Android ⓘ	✓	✓
Maven, Gradle, sbt	✓	✓
Git, SVN, Mercurial	✓	✓
Debugger	✓	✓
Profiling tools ⓘ	✓	✗
Spring, Java EE, Micronaut, Quarkus, Helidon, and more ⓘ	✓	✗
Swagger, Open API Specifications	✓	✗
JavaScript, TypeScript ⓘ	✓	✗
Database Tools, SQL	✓	✗

## Prebuilt OpenJDK Binaries for Free!

Java™ is the world's leading programming language and platform. AdoptOpenJDK uses [infrastructure](#), [build](#) and [test](#) scripts to produce prebuilt binaries from [OpenJDK™](#) class libraries and a choice of either [OpenJDK](#) or the [Eclipse OpenJ9](#) VM.

All AdoptOpenJDK binaries and scripts are [open source licensed](#) and available for free.

### Download for macOS x64

#### 1. Choose a Version

- ☐ OpenJDK 8 (LTS)  
☒ OpenJDK 11 (LTS)  
☐ OpenJDK 15 (Latest)

#### 2. Choose a JVM

[Help Me Choose](#)

- ☒ HotSpot  
☐ OpenJ9

[Download](#) Latest release

jdk-11.0.8+10

[Other platforms](#) ⓘ

[Release Archive & Nightly Builds](#) ⓘ

AdoptOpenJDK now also distributes [OpenJDK upstream builds!](#) (Built by Red Hat)



man kann fast alles in Java umsetzen

- einfach zu..... lesen :-)
- Objektorientiert
  - Klassen
  - Polimorphismus
  - Generics
  - strenge Typisierung
- Unabhängig von Plattform
  - Durch Übersetzung in Virtuelle Maschine (JVM)
- Reich an Libraries
  - Netzwerk
  - Nebenläufigkeit
  - Sicherheit
  - Web
  - ...

**JAVA**



**PYTHON**



**C++**





# Aspekte von Java

## Nachteile

- Laufzeithandicap durch Interpretation der JVM

## Vorteile

- Verteilte Anwendungen, Web-Anwendungen
- Rechnerunabhängigkeit
- einfach zu lernen
- einfach zu verstehen
- viele Libraries
- starke Community
- viele Jobs
- ...



## ein Java-Programm...

- Ein Java-Programm besteht aus
  - **eine Menge von Klassen und Schnittstellen**
- Eine Klasse besteht aus
  - **Attributen:** Beschreiben den Zustand eines Objekts
  - **Methoden:** Beschreiben die Operationen, die ein Objekt ausführen kann
  - **Konstruktoren:** Operationen zur Erzeugung von Objekten



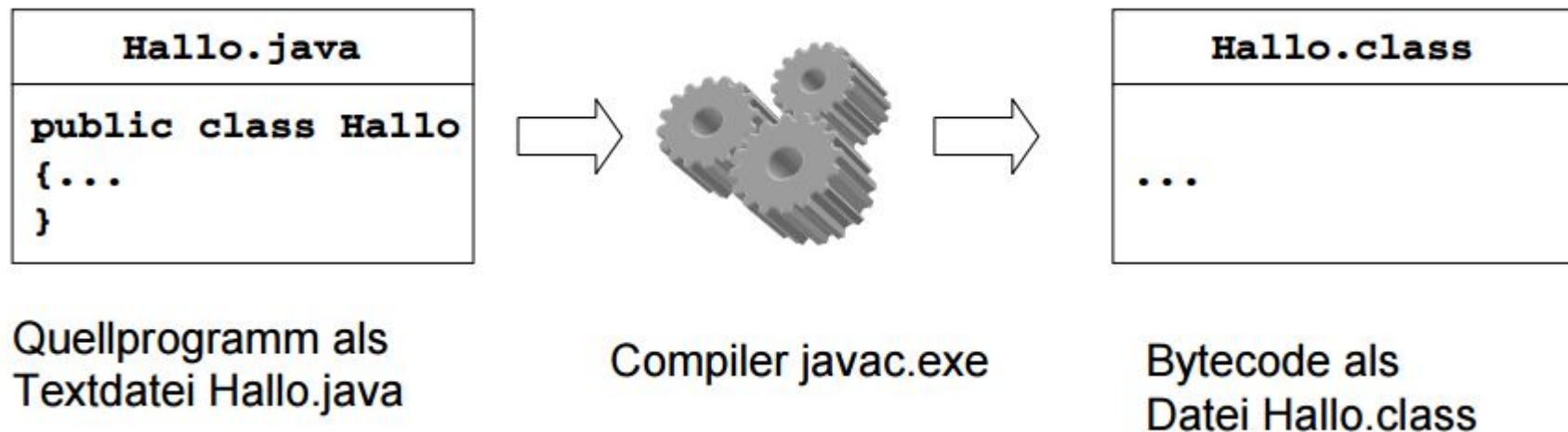
# Beispiel

```
public class Hallo {  
    public static void main(String[] args) {  
        System.out.println(„Hallo!“);  
    }  
}
```

- **Methodenaufruf (allgemein):**
  - `object.methodName(actual parameters);`
- **Beispiel:**
  - `System.out.println(„Hallo!“);`

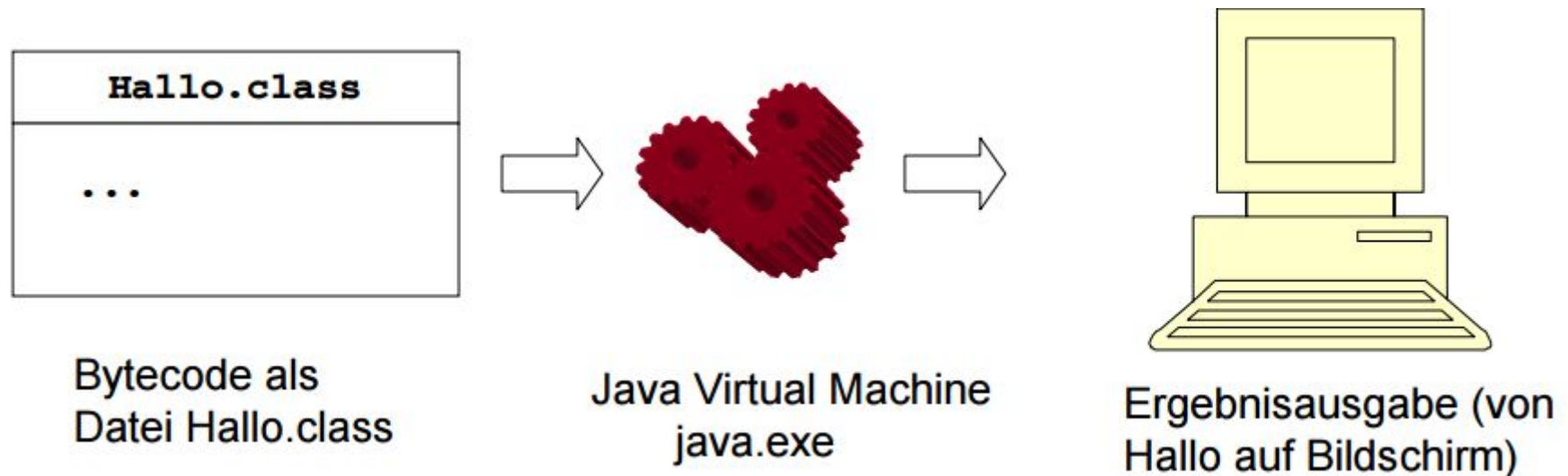
# Übersetzung und Ausführung

- Übersetzung in Bytecode
  - Aus einer Textdatei mit Endung „.java“ erzeugt der Compiler **javac** eine Datei mit gleichem Namen, aber Endung „.class“
  - Diese enthält den Bytecode für die JVM



# Übersetzung und Ausführung

Die Datei mit dem Bytecode wird der **JVM** übergeben und von der **JVM** ausgeführt (d.h. interpretiert).





# Übersetzung und Ausführung

## Übersetzung von Hallo.java:

```
cat@darkstar:~$ javac Hallo.java
```

## Interpretation von Hallo.class:

```
cat@darkstar:~$ java Hallo
```

## Gibt auf dem Bildschirm aus:

*Hello world!*

```
1  class Main {  
2      public static void main(String[] args) {  
3          System.out.println("Hello world!");  
4      }  
5  }  
6  
7
```

# Primitive Datentypen

- Wahrheitswerte

- `boolean`

- Zeichen

- `char`

- Zahlen

- `byte`
- `short`
- `int`
- `long`
- `float`
- `double`

Datentyp	Wertebereich	BIT
<code>boolean</code>	true, false	8
<code>char</code>	0 bis 65.535	16
<code>byte</code>	-128 bis 127	8
<code>short</code>	32.768 bis 32.767	16
<code>int</code>	-2.147.483.648 bis 2.147.483.647	32
<code>long</code>	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807	64
<code>float</code>	+/- 1,4E-45 bis +/- 3,4E+38	32
<code>double</code>	+/- 4,9E-324 bis +/- 1,7E+308	64



## Variablen und Konstanten

- Deklaration von Variablen:

<Datentyp> <Name>;

Beispiel:

```
boolean a;
```

- Zuweisung von Werten:

<Datentyp> <Name> = <Wert>;

Beispiel:

```
int b;  
b = 7;  
boolean a = true;  
char c,d,e;
```

# Variablen und Konstanten

- Beschränkungen für Variablenbezeichnungen:
  - beginnen mit einem Buchstaben oder Unterstrich
  - Nachdem dürfen aber sowohl Buchstaben als auch Zahlen folgen
  - Operatoren und Schlüsselwörter nicht erlaubt
- Reservierte Schlüsselwörter:

*abstract, assert, boolean, break, byte, case, catch, char, class, const, continue, default, do, double, else, enum, extends, false, final, finally, float, for, future, generic, goto, if, implements, import, inner, instanceof, int, interface, long, native, new, null, operator, outer, package, private, protected, public, rest, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, true, try, var, void, volatile, while*

# Variablen und Konstanten

- Konstanten:
  - Variablen, die während des Programmablaufs unverändert bleiben sollen
- Beispiel **PI** als Variable:

```
double pi = 3.14159;
```

- Deklaration von Konstanten:  
`final <Datentyp> <NAME>;`
- Beispiel **PI** als Konstante

```
final double pi = 3.14159;
```

# Kommentare

1. `//` entire line
2. `/*` multiple  
lines `*/`
3. `/**` used by documentation Javadoc tool  
multiple  
lines`*/`

```
/* ... */  
*/ ... */ NOT OK!
```

```
/* ...  
//  
//  
*/ OK!!
```

## Arrays und Matrizen

- Erzeugen eines int-Arrays mit k Elementen:

`<Datentyp>[] <name>;`

`<name> = new <Datentyp>[k];`

- Oder in einer Zeile:

`<Datentyp>[] <name> = new <Datentyp>[k];`

- Zugriff auf Elemente und Initialisierung der Variablen:

```
int[] a = new int[2];  
a[0] = 3;  
a[1] = 4;
```

## Arrays und Matrizen

- Wir erzeugen Matrizen, indem wir zum Array eine Dimension dazu nehmen:

```
int[][] a = new int[n][m];  
a[4][1] = 27;
```

- Auf diese Weise können wir sogar noch mehr Dimensionen erzeugen:

```
int[][][][] a = new int[k][l][m][n];
```

# Char

- Bezeichnet ein Zeichen oder Symbol

```
char d = '7';  
char e = 'b';
```

- Relationale Operatoren (Vergleichsoperatoren)

```
boolean d, e, f, g;  
char a, b, c;  
a = '1';  
b = '1';  
c = '5';  
d = a == b;  
e = a != b;  
f = a < c;  
g = c >= b;
```

# Int

- Der ganzzahlige Datentyp `int` wird wahrscheinlich von allen am häufigsten verwendet
- Der kleinste und größte darstellbare Wert existiert als Konstante

```
int a, b = 0;  
a = 10;
```

```
int minimalerWert = Integer.MIN_VALUE;  
int maximalerWert = Integer.MAX_VALUE;
```

```
int a = 29, b, c;  
b = a/10;  
c = a%10;
```

```
int d=0, e=1;  
d = d + e;  
e = e - 5;  
d = d + 1;
```

```
int d=0, e=1;  
d += e;  
e -= 5;  
d += 1;  
d++;
```





# float, double

- Repräsentieren gebrochene Zahlen oder Gleitkommazahlen

5.                      4.3                      .00000001                      -2.87

- Java verwendet eine wissenschaftliche Darstellung für Gleitkommazahlen, um längere Zahlen besser lesen zu können

1E-8

- Die möglichen Operationen sind die gleichen, wie sie bei int vorgestellt wurden
  - lediglich die beiden Teilungsoperatoren verhalten sich anders.

# Gültigkeitsbereich

- Der Gültigkeitsbereich einer lokalen Variablen oder Konstante ist der die Deklaration umfassende Block
- Außerhalb dieses Blocks existiert die Variable nicht!

```
1. {  
    int wert = 0;  
    wert = wert + 17;  
    1.1 {  
        int total = 100;  
        wert = wert - total;  
    }  
    wert = 2 * wert;  
}
```

Block 1.1  
Gült.ber.  
total

Block 1.  
Gült.ber.  
wert

## Verzweigungen

Wenn eine Bedingung erfüllt ist, dann führe eine einzelne Anweisung aus:

```
if (<Bedingung>) <Anweisung>;
```

```
if (<Bedingung>) {
```

```
    <Anweisung_1>;
```

```
    <Anweisung_2>;
```

```
    ...
```

```
    <Anweisung_n>;
```

```
}
```

```
if (<Bedingung>) <Anweisung_1>;
```

```
else <Anweisung_2>;
```

# Verzweigungen

Mehrfachverzweigungen lassen sich mit switch realisieren

```
switch (<Ausdruck>) {  
    case <Konstante1>:  
        <Anweisung1>;  
    break;  
    case <Konstante2>:  
        <Anweisung2>;  
    break;  
    default:  
        <Anweisung3>;  
}
```

## Beispiel für switch

```
for (int i=0; i<5; i++){  
    switch(i){  
        case 0:  
            System.out.println("0");  
            Break;  
  
        case 1:  
        case 2:  
            System.out.println("1 oder 2");  
            break;  
  
        case 3:  
            System.out.println("3");  
            break;  
  
        default:  
            System.out.println("hier landen alle anderen...");  
            break;  
    }  
}
```

## Verschiedene Schleifentypen

```
for (<Startwert>; <Bedingung>; <Schrittweite>)  
    <Anweisung>;
```

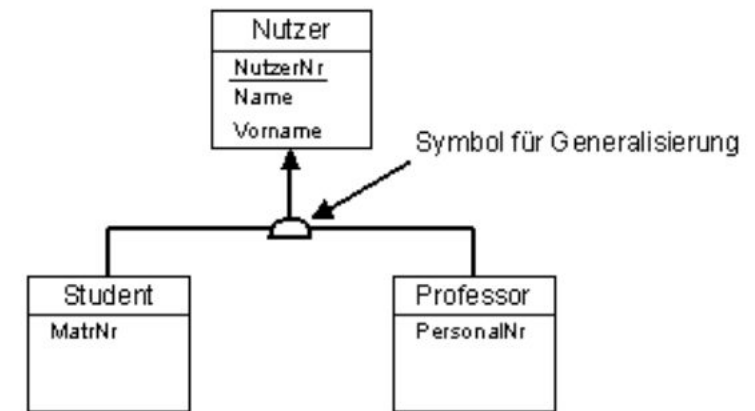
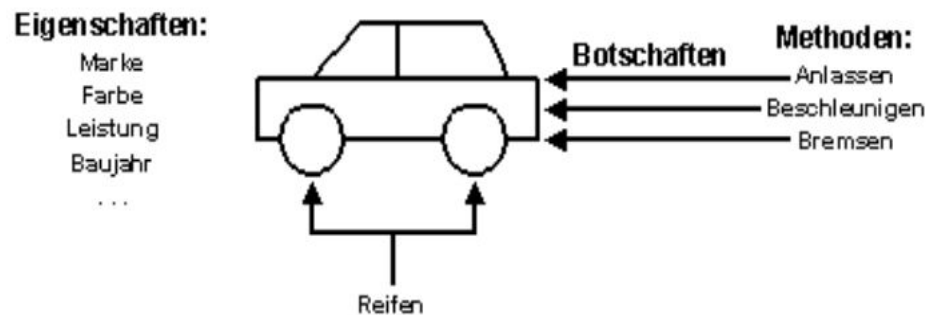
```
while (<Bedingung>)  
    <Anweisung>;
```

```
do {  
    <Anweisung>;  
} while (<Bedingung>) ;
```

- die Anweisungen wird mindestens einmal ausgeführt

# Objektorientierung

- Autos und Autoteile
- Professoren und Studenten
- ...



- ein abstraktes Modell
- bzw. ein Bauplan für eine Reihe von ähnlichen Objekten

## einfaches Beispiel

- Welche Eigenschaften hat jeder Würfel?

- Farbe
- Kantenlänge

- Welche Methoden hat jeder Würfel?

- Drehen
- Einfärben
- Zeichnen

```

1  public class Wuerfel {
2      private String farbe;
3      private int kantenlaenge;
4
5      public void drehen () {
6          System.out.println("Wuerfel::drehen()");
7      }
8
9      public void einfaerben () {
10         System.out.println("Wuerfel::einfaerben()");
11     }
12
13     public void zeichnen () {
14         System.out.println("Wuerfel::zeichnen()");
15     }
16 }
17

```

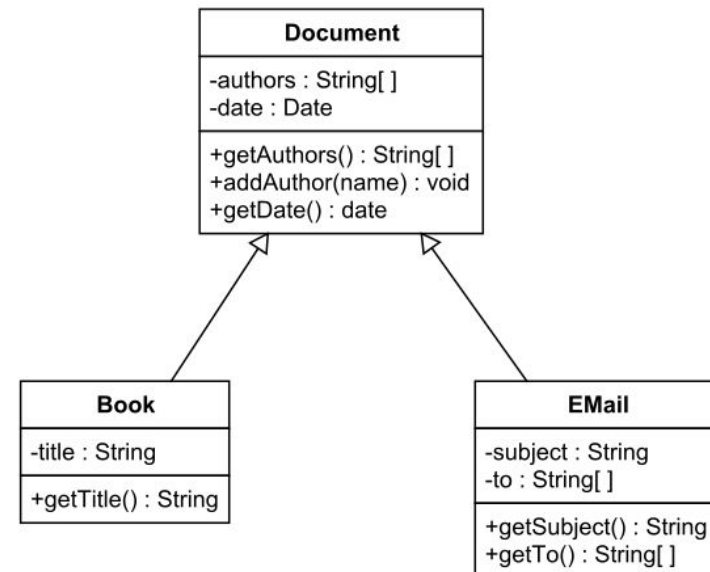
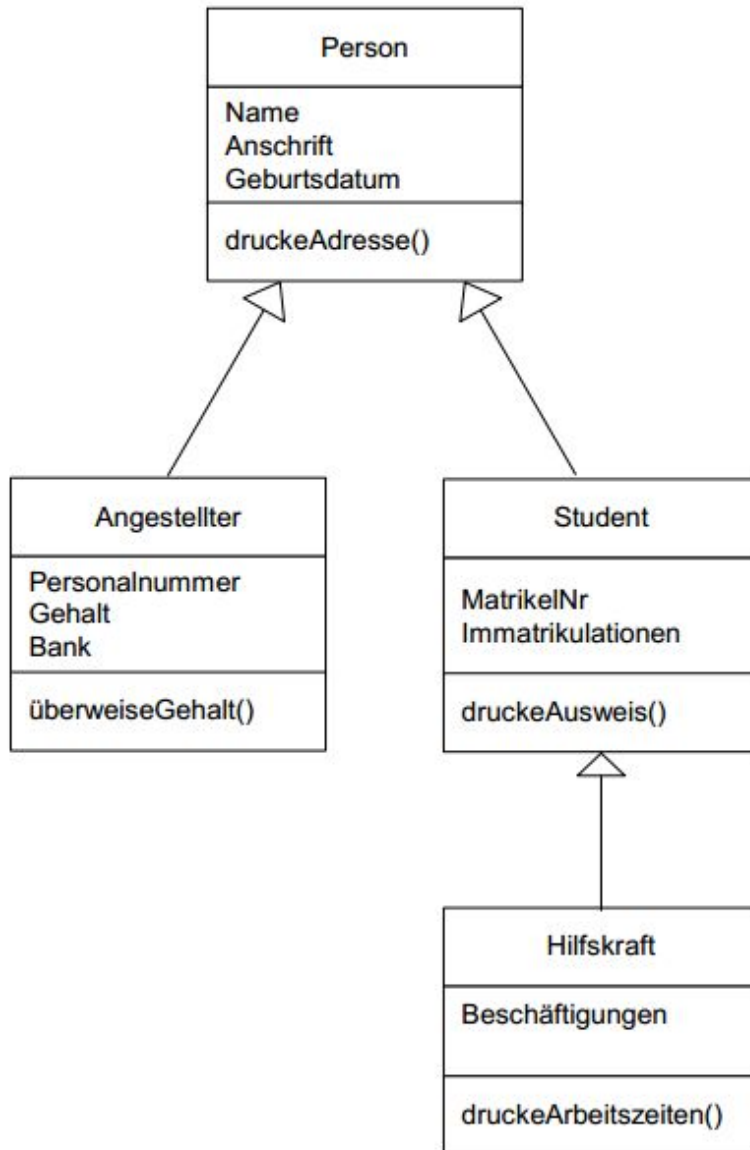




# Vererbung

- beschreibt eine Beziehung zwischen einer allgemeinen Klasse (**Basisklasse**) und einer **speziellen Klasse (Unterklasse)**
- die spezialisierte Klasse ist vollständig konsistent mit der Basisklasse, enthält aber **zusätzliche Informationen** (Attribute, Methoden, Assoziationen etc)
- ein Objekt der spezialisierten Klasse kann überall dort verwendet werden, wo ein Objekt der Basisklasse erlaubt ist

# Vererbung





# Vererbung

```
public class Person{  
    // Eigenschaften einer Person:  
    public String name;  
    public int alter;  
}
```

```
public class Spieler extends Person{  
    // Zusätzliche Eigenschaften eines Spielers:  
    public int staerke; // von 1 (schlecht) bis 10 (super)  
    public int torschuss; // von 1 (schlecht) bis 10 (super)  
    public int motivation; // von 1 (schlecht) bis 10 (super)  
    public int tore;  
}
```



# Liskovisches Prinzip

```
class FormaGeometrica {  
    public double calculArea() {  
        return 0.0;  
    }  
}
```

```
class Dreptunghi extends  
FormaGeometrica {  
    private double lungime;  
    private double latime;  
  
    public Dreptunghi(double  
lungime, double latime) {  
        this.lungime = lungime;  
        this.latime = latime;  
    }  
  
    @Override  
    public double calculArea() {  
        return lungime * latime;  
    }  
}
```

```
class Cerc extends FormaGeometrica {  
    private double raza;  
  
    public Cerc(double raza) {  
        this.raza = raza;  
    }  
  
    @Override  
    public double calculArea() {  
        return Math.PI * raza * raza;  
    }  
}  
  
public class LSPDemo{  
    public static void main(String[] args) {  
        FormaGeometrica dreptunghi = new  
Dreptunghi(5, 4);  
        FormaGeometrica cerc = new Cerc(3);  
  
        aria(dreptunghi);  
        aria(cerc);  
    }  
  
    public static void aria(FormaGeometrica  
forma) {  
        System.out.println(forma.calculArea());  
    }  
}
```



# Liskovisches Prinzip

```
public class Rectangle {  
  
    private int length;  
    private int breadth;  
  
    public int getLength() {  
        return length;  
    }  
    public void setLength(int  
length) {  
        this.length = length;  
    }  
    public int getBreadth() {  
        return breadth;  
    }  
    public void setBreadth(int  
breadth) {  
        this.breadth = breadth;  
    }  
    public int getArea() {  
        return this.length *  
this.breadth;  
    }  
}
```

```
public class Square extends Rectangle {  
  
    @Override  
    public void setBreadth(int breadth) {  
        super.setBreadth(breadth);  
        super.setLength(breadth);  
    }  
    @Override  
    public void setLength(int length) {  
        super.setLength(length);  
        super.setBreadth(length);  
    }  
}  
  
    public class LSPDemo {  
  
        public void calculateArea(Rectangle r) {  
            r.setBreadth(2);  
            r.setLength(3);  
            r.getArea();  
        }  
  
        public static void main(String[] args) {  
            LSPDemo lsp = new LSPDemo();  
            lsp.calculateArea(new Rectangle());  
            lsp.calculateArea(new Square());  
        }  
    }
```



# Methoden

- Objekte interagieren miteinander durch das gegenseitige Aufrufen von Methoden (**Nachrichtenaustausch**).
- Eine Methode (**Nachricht**) besteht aus zwei Teilen:
  - **die Signatur**, die Angaben über Sichtbarkeit, Rückgabebetyp, Name der Methode und Parameter macht.
  - der Methodenrumpf, in dem die Deklarationen der lokalen Variablen und die eigentlichen Anweisungen stehen.

# Methoden

```
class Point {  
    public double x, y;  
    //Methodendeklaration:  
    public double distance(Point pkt) {  
        double xdiff = x - pkt.x;  
        double ydiff = y - pkt.y;  
        return Math.sqrt(xdiff*xdiff +ydiff*ydiff);  
    }  
    ...  
    //Methodenaufruf:  
    Class PointTester {  
        public static void main(String[] args) {  
            Point lowerLeft = new Point(); lowerLeft.x = 0.1; ...  
            Point upperRigth = new Point(); upperRight.x = 0.1; ...  
            double d = lowerLeft.distance(upperRight);  
        }  
    }
```



## Standardvariable „this“

- `this` ist eine Referenz zum aktuellen Objekt
- oder anders ausgedrückt `this` ist eine Referenz auf die aktuelle Instanz der Klasse in der man sich gerade befindet
- Über `this` kann auf alle Variablen und Methoden der Instanz zugegriffen werden
- für constructor chaining
  - muss erste Anweisung im Konstruktor sein
- um Ambiguities zu vermeiden (zB setters)





# Statische Elemente

- Variablen und Methoden, die nicht zu einer bestimmten Instanz sonder zur Klasse gehören
- Statische Variablen/Methoden sind auch dann verfügbar, wenn noch keine Instanz der Klasse erzeugt wurde
- Statische Variablen/Methoden können über den Klassennamen aufgerufen werden
- Deklaration durch das Schlüsselwort: `static`

```
class Point {  
    double x, y;  
    static int count;  
}
```



# Konstrukturen

- Jede Klasse benötigt **einen oder mehrere** Konstrukturen, welche:
  - reservieren den Speicherplatz für eine neue zu erzeugende Instanz
  - weisen den Instanzvariablen initiale Werte zu
  - haben denselben Namen wie die Klasse
  - werden wie Methoden deklariert, aber ohne Rückgabewert

# Konstrukturen

Wenn kein Konstruktor erstellt wurde, wird von Java default-mäßig ein Konstruktor (**ohne Parameter**) zur Verfügung gestellt.

```
class Student {  
    private String matrNr;  
    private String name;  
    private int semester;  
  
    Student(String name, String matrNr) {  
        this.name = name;  
        this.matrNr = matrNr;  
    }  
}
```



# Spezialisierte Konstruktoren

```
class Student {  
    private String name, matrNr;  
    private int semester;  
  
    Student(String studName, String studMatrNr) {  
        //Konstruktor 1  
        name = studName;  
        matrNr = studMatrNr;  
    }  
  
    Student(String name, String matrNr, int semester) {  
        // Konstruktor 2  
        this(name, matrNr); //Aufruf Konstruktor 1  
        this.semester=semester;  
    }  
}
```



# Zugriffskontrolle

- In JAVA gibt es drei Attribute und eine Standardeinstellung, die den Gültigkeitsbereich von Klassen, Variablen und Methoden regeln, d.h. festlegen, ob bzw. welche anderen Objekte auf eine Klasse, Variable oder Methode der programmierten Klasse zugreifen können:
  - private
  - protected
  - public



# JUnit

- Motivation
- Extreme Programming
- Test-First
- Das Framework
  - Grundlagen
  - Assert
  - TestCase



# Motivation

- JUnit: Open source Test-Framework
  - Schreiben und Ausführen automatischer Unit Tests unter Java
  - Aktuell: Version 5 (<http://www.junit.org/>)
  - In jeder gängigen Java-IDE verwendbar
- Autoren:
  - Kent Beck (Extreme Programming)
  - Erich Gamma
- Entsprechende Frameworks für gängige Programmiersprachen erhältlich

# TestFirst-Ansatz

- Ziel:
  - Qualitätssicherung: Testbarkeit, Einfachheit
  
- Vorgehen:
  - Vor eigentlicher Codierung Test schreiben
  - Erst wenn Test fehlerfrei, dann ist Code fertig
  - Nur soviel Produktionscode wie Test verlangt
  - Kleine Schritte (abwechselnd Test- und Produktionscode)
  - Vor Integration in Gesamtsystem muss Unit Test erfolgreich sein





# TestFirst-Ansatz

- Vorteile:
  - gesamter Code ist getestet (Zerstörung funktionierenden Codes sofort entdeckt)
  - Tests dokumentieren Code
  - Schnelles Feedback durch kurze Wechsel Erzeugung von Test- und Produktionscode
  - Einfaches Design, dadurch Test bestimmtes Design
- Vorsicht:
  - **triviale Testfälle**

# JUnit Beispiel

```
class Calculator {  
    public double sum(double a, double b)  
    public double diff(double a, double b)  
    public double mult(double a, double b)  
    public double div(double a, double b)  
  
    public void setMem(double a)  
  
    public double getMem()  
  
    public void clearMem()  
}
```

```
1 public class CalculatorTest {  
2     private Calculator calculator;  
3  
4     @BeforeEach public void setup() {  
5         calculator = new Calculator();  
6     }  
7  
8     @Test public void test_sum () {  
9         double a = 1.2, b = 2.3;  
10  
11         assertEquals(3.5, calculator.sum(a,b), 0.1);  
12     }  
13  
14     @Test (expected = IllegalArgumentException.class)  
15     public void test_div () {  
16         double a = 1.2, b = 0.0;  
17  
18         calculator.div(a,b);  
19     }  
20 }  
21
```

# JUnit Beispiel

- decorators für die Methoden der Test-Klasse

- `@BeforeAll/@BeforeEach`
- `@Test`
- `@AfterAll/@AfterEach`

- `@TestFactory` – denotes a method that is a test factory for dynamic tests
- `@DisplayName` – defines custom display name for a test class or a test method
- `@Nested` – denotes that the annotated class is a nested, non-static test class
- `@Tag` – declares tags for filtering tests
- `@ExtendWith` – it is used to register custom extensions
- `@BeforeEach` – denotes that the annotated method will be executed before each test method (previously `@Before`)
- `@AfterEach` – denotes that the annotated method will be executed after each test method (previously `@After`)
- `@BeforeAll` – denotes that the annotated method will be executed before all test methods in the current class (previously `@BeforeClass`)
- `@AfterAll` – denotes that the annotated method will be executed after all test methods in the current class (previously `@AfterClass`)
- `@Disable` – it is used to disable a test class or method (previously `@Ignore`)