

# Programmieren II: Java

## Grundlagen der Objektorientierung in Java

Prof. Dr. Christopher Auer

Sommersemester 2024



Objektorientierung und UML

Klassen, Objekte und Referenzen

Konstruktoren

Datenkapselung

Unveränderliche Klasse

Klassenvariablen und -Methoden

Enumerationen

Kopieren

Identität und Gleichheit

Dokumentation mit javadoc

# Inhalt

## Objektorientierung und UML

Warum Objektorientierung?

UML

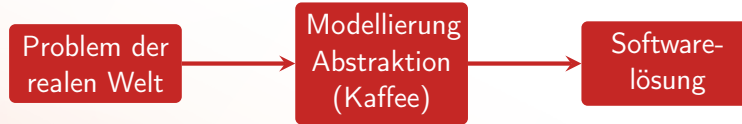
# Inhalt

## Objektorientierung und UML

Warum Objektorientierung?

# Warum Objektorientierung?

- ▶ Aufgabe eines Softwareentwicklers



- ▶ Probleme der realen Welt **bestehen** aus...
  - ▶ **Objekten**: Personen, Produkte, Prozesse, ...
  - ▶ **Beziehungen zwischen Objekten**: Person „arbeitet in“ Organisation, Produkt „besteht aus“ Teilen, Prozess „besteht aus“ Schritten, ...
  - ▶ **Verhalten von Objekten**: Person „verlässt“ Organisation, Produkt „wird erstellt“, Prozess „wird durchgeführt“
- ▶ Modellierung in **prozeduralen Programmiersprachen** (z.B. C) durch
  - ▶ (Einfache) Datenstrukturen
  - ▶ Unterprogramme
- ▶ Abbildung von Problemen der realen Welt ist schwierig

# Darum Objektorientierung!

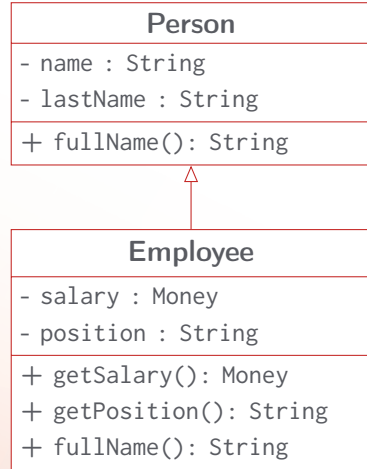
- ▶ Zentrales Element der Objektorientierten Programmierung (OOP) ist das **Objekt**, es hat...
  - ▶ eine **Identität**: ist eindeutig und unveränderlich
  - ▶ einen **Zustand**: Attribute und Beziehungen zu anderen Objekten
  - ▶ ein **Verhalten**: das z.B. den Zustand verändern kann
- ▶ Eine **Klasse** ist eine **Schablone** („**template**“) für Objekte mit
  - ▶ gleicher Zusammensetzung an **Attributen**
  - ▶ gleicher Definition des **Verhaltens**

Person
- name : String - lastName : String
+ getFullName(): String

- ▶ Objekte einer Klasse nennt man auch **Instanzen** („**instances**“)

# Darum Objektorientierung!

- ▶ OOP ermöglicht das **direkte Modellieren** von Dingen der realen Welt
- ▶ **Wichtige Eigenschaften** der OOP
  - ▶ **Kapselung**: Daten eines Objekts können nur über Schnittstelle (Methoden) verändert werden
  - ▶ **Vererbung**: Klassen und ihr Verhalten können spezialisiert werden (Wiederverwendbarkeit)
  - ▶ **Polymorphie**: Gleiche Schnittstelle, führt je nach dahinterliegender Implementierung, zu unterschiedlichem Verhalten



# Inhalt

## Objektorientierung und UML

### UML



# UML

- ▶ UML: „Unified Modeling Language“
- ▶ Vereinheitlichte (grafische) **Modellierungssprache** zur Softwareentwicklung
- ▶ Diagrammtypen (Auswahl)
  - ▶ **Klassendiagramme**: Klassen und ihre Beziehungen
  - ▶ **Use-Case-Diagramme**: Interaktion von Nutzer mit Software
  - ▶ **Sequenzdiagramme**
- ▶ Hier vor allem **Klassendiagramme**

Klassenname
- attribut1 : Typ1 - attribut2 : Typ2
+ methode1(argA : TypA): TypB + methode2(): <b>void</b>

Person
- name : String - father : Person - mother : Person
+ getFather(): Person + getMother(): Person + talk(): String

# UML: Instanzen (Objekte)

## tywin: Person

- name = "Lannister, Tywin"
- father = **null**
- mother = **null**

- + getFather(): Person
- + getMother(): Person
- + talk(): String

## cersei: Person

- name = "Lannister, Cersei"
- father = tywin
- mother = joanna

- + getFather(): Person
- + getMother(): Person
- + talk(): String

## jaime: Person

- name = "Lannister, Jaime"
- father = tywin
- mother = joanna

- + getFather(): Person
- + getMother(): Person
- + talk(): String

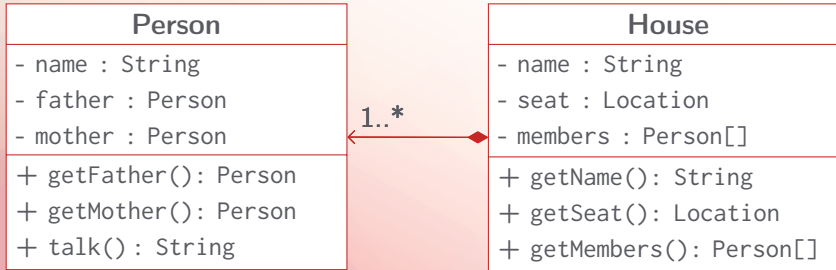
## joffrey: Person

- name = "Lannister, Joffrey"
- father = jaime
- mother = cersei

- + getFather(): Person
- + getMother(): Person
- + talk(): String

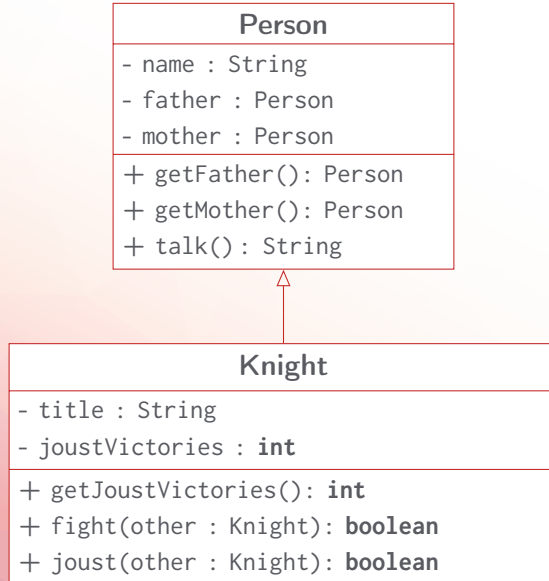
# Assoziation, Aggregation, Komposition

- ▶ Assoziationen beschreiben die **Beziehungen zwischen Objekten**
- ▶ Für „besteht aus“-Beziehungen:
  - ▶ **Aggregation**: Teile können für sich existieren (Reifen am Auto)
  - ▶ **Komposition**: Teile machen nur in der Komposition Sinn (Räume in Gebäuden)



# Vererbung

Vererbung wird durch einen **weißen Pfeil** dargestellt



# Inhalt

## Klassen, Objekte und Referenzen

- Klassen

- Referenzen und Instanzen

- Attribute einer Klasse: Objektvariablen

# Inhalt

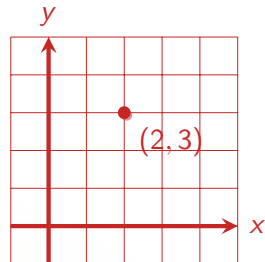
## Klassen, Objekte und Referenzen

### Klassen

## Beispiel: Point2D

- Point2D modelliert einen Punkt im  $\mathbb{Z}^2$

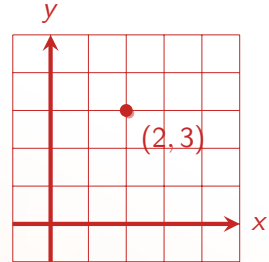
Point2D
<ul style="list-style-type: none"><li>- x : <b>int</b></li><li>- y : <b>int</b></li></ul>
<ul style="list-style-type: none"><li>+ Point2D(x : <b>int</b>, y : <b>int</b>)</li><li>+ Point2D()</li><li>+ Point2D(other : Point2D)</li><li>+ getX() : <b>int</b></li><li>+ setX(x : <b>int</b>)</li><li>+ getY() : <b>int</b></li><li>+ setY(y : <b>int</b>)</li><li>+ set(x : <b>int</b>, y : <b>int</b>)</li><li>+ move(dx : <b>int</b>, dy : <b>int</b>)</li><li>+ distance(p : Point2D): <b>double</b></li></ul>



- Implementierung: `shapes/Point2D.java`

# Bestandteile der Klasse

- ▶ **Name:** Point2D
- ▶ **Attribute:**  $x$ - und  $y$ -Koordinate
- ▶ **Operationen:**
  - ▶ **Konstruktoren:** Initialisieren Objekt
  - ▶ **Getter/Setter:** Modifizieren Attribute
  - ▶ **Abfragen:** z.B. Distanz-Berechnung
- ▶ Point2D stellt Prototypen für **konkrete** Punkte dar
- ▶ Wie **verwendet** man Point2D?





# Inhalt

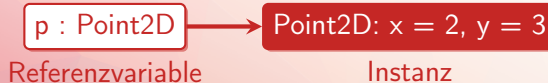
## Klassen, Objekte und Referenzen

### Referenzen und Instanzen


# Instanzen und Referenzen

```
Point2D p = new Point2D(2,3);
```

- ▶ **new-Operator**
  - ▶ **Unärer Operator** mit Klassenname als Argument
  - ▶ **Operation**
    1. Reserviert **Speicher** und erstellt **Instanz/Objekt**
    2. Ruft passenden **Konstruktor** auf
    3. Gibt **Referenz** auf erstelltes Objekt zurück
- ▶ Referenz Point2D p
- ▶ **Zur Erinnerung:** Referenz beinhaltet
  - ▶ Zeiger auf Speicherbereich
  - ▶ Typinformationen
- ▶ p ist **Referenzvariable** die auf erstellte Instanz **verweist**



## Beispiel: Referenzen

```
1  runReferencesExample
2 Point2D p1, p2, p3;
3 p1 = new Point2D(0,0);
4 p2 = new Point2D(0,1);
6 System.out.printf("Distanz: %f\n", p1.distance(p2));
8 p3 = p2;
9 p3.set(1,1);
11 System.out.printf("Distanz: %f\n", p1.distance(p2));
```

 References.java

Zeile 8

p1 : Point2D

Point2D: x = 0, y = 0

p2 : Point2D

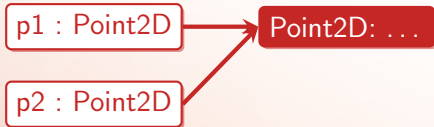
Point2D: x = 1, y = 1

p3 : Point2D

# Referenzen vs. primitive Typen

```
Point2D p1, p2;  
p1 = new Point2D(1,2);  
p2 = p1;
```

```
double x, y;  
x = 3.1415;  
y = x;
```




x = 3.1415

y = 3.1415

- ▶ Bei Zuweisungen wird der **Inhalt** kopiert
  - ▶ **Primitive Typen**: Wert (z.B. 3.1415)
  - ▶ **Referenzen**: Verweis auf die Instanz
- ▶ Zwei Referenzen sind **gleich**, wenn sie auf die **dieselbe Instanz** verweisen

```
p1 = p2;  
if (p1 == p2 ) // true  
    // ...
```

# Vergleichen von Referenzvariablen

```
37  runReferenceEqualityExample2
38 Point2D p1 = new Point2D(1,2);
39 Point2D p2 = new Point2D(1,2);
40 System.out.printf("p1 == p2: %b%n", p1 == p2);
41 p2 = p1;
42 System.out.printf("p1 == p2: %b%n", p1 == p2);
```

 References.java

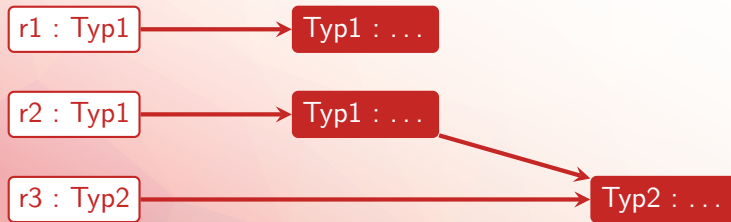
Frage: Was ist die **Ausgabe**?

```
p1 == p2: false
p1 == p2: true
```

- ▶ **Erste Ausgabe** ( $p1 \neq p2$ )
  - ▶  $p1$  und  $p2$  verweisen auf **unterschiedliche** Instanzen
  - ▶ Attribute der Instanzen sind (zufälligerweise) **wertgleich**
- ▶ **Zweite Ausgabe** ( $p1 == p2$ )
  - ▶  $p1$  und  $p2$  verweisen auf die **dieselbe** Instanz
- ▶ **Später**: Wertvergleich von Instanzen

# Einschub: Speicherverwaltung in Java

- ▶ Bei **new** reserviert JVM Speicher im „heap“
- ▶ **Kein** free/delete und **keine** Desktrutoren
- ▶ Wie wird Speicher wieder **freigegeben**?
- ▶ „Garbage Collector“ (GC)
  - ▶ Gibt Speicher **nicht mehr referenzierter** Objekte frei
  - ▶ Wird **automatisch** von JVM ausgeführt
  - ▶ Expliziter Aufruf mit ↗ `System.gc();`
- ▶ Beispiel



# Spezielle Referenzen

## ▶ null

- ▶ **Typenlos**: Kann jeder Referenzvariable zugewiesen werden
- ▶ Erzeugt bei Zugriffen eine [NullPointerException](#)
- ▶ z.B. zur **Initialisierung von Referenzvariablen**
- ▶ **Beispiel**

```
Point2D p = null;  
p.setX(0);
```

Fehler: [NullPointerException](#)

## ▶ this

- ▶ Referenz auf **aktuelles Objekt** (meist optional)
- ▶ **Beispiel**

```
public void move(int dx, int dy){  
    this.x += dx;  
    this.y += dy;  
}
```

- ▶ **super**: Referenz auf **Instanz der Basisklasse** (später)

# Zugriff auf Attribute und Methoden

- ▶ Punkt-Operator (schon oft verwendet)
- ▶ Methodenzugriff

```
var p = new Point2D(0,0);  
p.move(1,1);
```

## ▶ Attributzugriff

- ▶ Neue Methode in Point2D

```
public void move(Point2D other){  
    this.x += other.x;  
    this.y += other.y;  
}
```

- ▶ Hinweis: x/y sind **private**, Zugriff in Point2D aber möglich
- ▶ Prinzipiell auch **schreibender Zugriff** auf other möglich

```
public void evilMove(Point2D other){  
    other.x = (int)(Math.random()*1000); // muahahaha...  
    other.y = (int)(Math.random()*1000);  
}
```



# Inhalt

## Klassen, Objekte und Referenzen

Attribute einer Klasse: Objektvariablen

# Objektvariablen

## ► Attribute der Klasse Point2D

```
10 private int x;  
11 private int y;
```

shapes/Point2D.java

## ► Zugriff in Methoden, wie auf lokale Variablen

```
92 public void move(final int dx, final int dy){  
93     x += dx;  
94     y += dy;  
95 }
```

shapes/Point2D.java

## ► Zugriff über **this** (meist optional, hier nicht)

```
69 public void setX(final int x) {  
70     this.x = x;  
71 }
```

shapes/Point2D.java

# Objektvariablen vs. lokale Variablen

	Objektvariablen	Lokale Variablen
Speicherort	Heap	Stack
Sichtbarkeit	Modifier/Block	Block
Lebensdauer	Objekt	Methode/Block
Initialwert	definiert	nicht definiert

Datentyp	Initialwert
<b>boolean</b>	<b>false</b>
Numerisch	0
<b>char</b>	u0000
Referenz	<b>null</b>

# Verschattung von Objektvariablen

- ▶ **Achtung:** Variablennamen können **verschattet** werden

- ▶ Durch **Parameternamen**

```
69 public void setX(final int x) {  
70     this.x = x;  
71 }
```

shapes/Point2D.java

Eindeutigkeit durch **this**

- ▶ Durch **lokale Variablen**

```
public void setX(int newX){  
    int x;  
    x = newX;  
}
```

- ▶ Lokalen Variablen **verschattet** Objektvariable
- ▶ Objektvariable bleibt **unverändert**
- ▶ Besser **this** verwenden: **this.x** = newX;
- ▶ Noch besser: anderen Bezeichner wählen

# Inhalt

## Konstrukturen

Initialisierung und Default-Werte

Verkettung von Konstrukturen

Arten von Konstrukturen

# Aufgaben und Definition eines Konstruktors

- ▶ Ein Konstruktor
  - ▶ bringt ein neu erstelltes Objekt in einen **initialen, gültigen Zustand**
  - ▶ kann über **Parameter gesteuert** werden
- ▶ Deklaration wie eine Methode **ohne Rückgabeparameter**

```
18 public Point2D() {  
19     this(0,0);  
20 }
```

shapes/Point2D.java

- ▶ Mit **Parametern**

```
35 public Point2D(final int x, final int y){  
36     set(x, y);  
37 }
```

shapes/Point2D.java

# Default-Konstruktor

- ▶ Ist **kein Konstruktor** angegeben, implementiert Java einen **Default-Konstruktor**
  - ▶ Keine **Parameter**
  - ▶ Keine **Anweisungen**
- ▶ **Beispiel**

```
public class VeryEmptyClass{  
}
```

beinhaltet implizit

```
public class VeryEmptyClass{  
    public VeryEmptyClass(){  
    }  
}
```

- ▶ In diesem Fall behalten Objektvariablen ihre **Default-Werte**
- ▶ Was ist wenn **andere Default-Werte** gewünscht sind?

# Inhalt

## Konstrukturen

Initialisierung und Default-Werte



# Initialisierung und Default-Werte

- Default-Werte können **überschrieben** werden:

```
4 public class Greeter{  
5     private String target = "World";  
7     public Greeter(){ }  
9     public Greeter(String target){  
10         this.target = target;  
11     }  
13     public void greet(){  
14         System.out.printf("Hello %s!\n", target);  
15     }  
16 }
```

Greeter.java

# Initialisierung und Default-Werte

- ▶ Auch **komplexere Ausdrücke** und **Methodenaufrufe** erlaubt

```
public class Greeter{  
    private String target = System.getenv("USERNAME");  
    /* ... */  
}
```

- ▶ **Oder** (unschön):

```
public class Greeter{  
    private String target =  
        (new Scanner(System.in)).nextLine();  
    /* ... */  
}
```

- ▶ **Frage:** Wann wird der Code ausgeführt?

## Experiment I

```
4 public class NumberPrinter
5 {
6     private double number = getRandomNumber();
7
8     public NumberPrinter(){
9         System.out.println("NumberPrinter()");
10    }
11
12    public NumberPrinter(double number){
13        System.out.printf("NumberPrinter(%f)%n", number);
14        this.number = number;
15    }
16
17    private double getRandomNumber(){
18        System.out.println("getRandomNumber()");
19        return 1000*Math.random();
20    }
21
22    public void printNumber(){
23        System.out.printf("Number: %f%n", number);
```


## Experiment II

```
24     }
```

```
26 }
```

NumberPrinter.java

## ► Konstruktor NumberPrinter()

```
6  runNumberPrinterExample1  
7 var numberPrinter = new NumberPrinter();  
8 numberPrinter.printNumber();
```


 Constructors.java

## ► Ausgabe

```
getRandomNumber()  
NumberPrinter  
Number: 681,660248
```

# Ergebnisse

- ▶ Konstruktor `NumberPrinter(double number)`

```
14  runNumberPrinterExample2  
15 var numberPrinter = new NumberPrinter(3.1415f);  
16 numberPrinter.printNumber();
```

 Constructors.java

- ▶ Ausgabe

```
getRandomNumber()  
NumberPrinter(3,141500)  
Number: 3,141500
```

- ▶ **Ergebnis:** Initialisierung wird **immer** vor dem Konstruktor aufgerufen

# Initializer

- ▶ **Initializer**: Alternative zu Initialisierung bei Deklaration
- ▶ **Namenloser Block** neben Attributen und Methoden

```
public class NumberPrinter{  
    private double number;  
    // Initializer  
    {  
        number = 1000 * Math.random();  
    }  
    /* ... */  
}
```

- ▶ Wird ebenfalls **vor dem Konstruktor** ausgeführt
- ▶ Zur **Übersichtlichkeit** bei komplexeren Initialisierungen

# Inhalt

## Konstrukturen

### Verkettung von Konstrukturen



# Verkettung von Konstruktoren

- ▶ Konstruktoren können **Konstruktoren** über **this** aufrufen
- ▶ **Beispiel** Point2D
  - ▶ Konstruktor mit **Initialwerten**

```
35 public Point2D(final int x, final int y){  
36     set(x, y);  
37 }
```

shapes/Point2D.java

- ▶ Konstruktor mit **Default-Werten**

```
18 public Point2D() {  
19     this(0,0);  
20 }
```

shapes/Point2D.java

- ▶ Konstruktor mit **anderem Punkt**

```
24 public Point2D(Point2D other){  
25     this(other.getX(), other.getY());  
26 }
```

shapes/Point2D.java

# Verkettung von Konstruktoren

- ▶ Welcher Konstruktor wird aufgerufen?
- ▶ Gleiche Regeln wie bei Überladung von Methoden
- ▶ Achtung: Aufruf von Konstruktor muss erste Anweisung sein
- ▶ Nicht erlaubt

```
public Point2D(){  
    System.out.println("Hello");  
    this(0,0);  
}
```

- ▶ Grund: Andere Konstrukteure können Konstruktor der Basisklasse aufrufen
- ▶ ...diese müssen immer zuerst ausgeführt werden

# Inhalt

## Konstrukturen

### Arten von Konstrukturen

# Arten von Konstruktoren

- ▶ **Default-Konstruktor:** Automatisch generiert wenn kein Konstruktor definiert
- ▶ **Copy-Konstruktor:** Kopiert Objekt gleichen Typs

```
24 public Point2D(Point2D other){  
25     this(other.getX(), other.getY());  
26 }
```

shapes/Point2D.java

- ▶ **Custom-Konstruktor:** „alle anderen“

```
35 public Point2D(final int x, final int y){  
36     set(x, y);  
37 }
```

shapes/Point2D.java

# Inhalt

## Datenkapselung

Beispiel: Die Klasse SimpleRectangle

Allgemeines Konzept: Schnittstellenvertrag

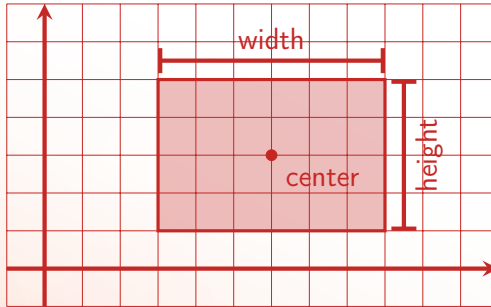
# Inhalt

## Datenkapselung

Beispiel: Die Klasse SimpleRectangle

# Die Klasse SimpleRectangle

- Die Klasse SimpleRectangle modelliert Rechtecke



- UML

SimpleRectangle	
+ center : Point2D	
+ width : <b>int</b>	
+ height : <b>int</b>	
+ getArea(): <b>int</b>	
+ containsPoint(other : Point2D): ↵	
boolean	

## Beispiel: Die Klasse SimpleRectangle


```
5 public class SimpleRectangle {  
6     public Point2D center;  
7     public int width;  
8     public int height;  
10    public int getArea(){  
11        return width * height;  
12    }  
14    public boolean contains(Point2D point){  
15        int deltaX = point.getX() - center.getX();  
16        int deltaY = point.getY() - center.getY();  
17        return Math.abs(deltaX) <= width/2  
18            && Math.abs(deltaY) <= height/2;  
19    }  
20 }
```

shapes/SimpleRectangle.java



# Verwendung von SimpleRectangle

## ► Beispiel 1:


```
7  runNegativeWidthExample  
8 SimpleRectangle rect = new SimpleRectangle();  
9 rect.width = 10;  
10 rect.height = -5;  
11 System.out.printf("Flaeche: %d%n", rect.getArea());
```

 shapes/SimpleRectangleExamples.java

Flaeche: -50

# Verwendung von SimpleRectangle

## ► Beispiel 2:

```
17  runNullCenterExample  
18 SimpleRectangle rect = new SimpleRectangle();  
19 rect.width = 10;  
20 rect.height = 5;  
21 System.out.printf("Beinhaltet (0,0): %b%n",  
22     rect.contains(new Point2D(0,0)));
```

 shapes/SimpleRectangleExamples.java

java.lang.NullPointerException

# Datenkapselung

- ▶ Das Beispiel zeigt: Offener Zugriff auf Attribute verursacht
  - ▶ inkonsistente Zustände
  - ▶ Fehler
- ▶ Datenkapselung (auch Geheimnisprinzip)
  - ▶ Attribute gehen nur Klasse/Objekt etwas an
  - ▶ Zugriff erfolgt ausschließlich (indirekt) über Methoden
  - ▶ Denn: Methodenaufruf erhält konsistenten Zustand
- ▶ Weitere Vorteile
  - ▶ Attribute unabhängig von Schnittstelle
  - ▶ Attribute können geändert werden ohne Änderung der Schnittstelle

# Geheimnisprinzip in Java

Wie wird das Geheimnisprinzip in **Java** realisiert?

- ▶ **Konstruktoren/Initialisierung** stellen **gültigen Initialzustand** her (schon gesehen)
- ▶ **private** — schützt vor unerlaubtem Zugriff anderer Klassen
- ▶ **Getter/Setter** für kontrollierten Zugriff
- ▶ **Unveränderliche Klassen** (später)

# Geheimnisprinzip in Java

- ▶ Veränderbare Attribute sind **nie public**
  - ▶ meist **private**: Zugriff nur von Klasse
  - ▶ seltener **protected**: Zugriff nur in Hierarchie
  - ▶ fast nie **Paket-sichtbar**: Zugriff innerhalb des Pakets

SimpleRectangle
<ul style="list-style-type: none"><li>– center : Point2D</li><li>– width : <b>int</b></li><li>– height : <b>int</b></li></ul>
<ul style="list-style-type: none"><li>+ SimpleRectangle(center : ↵ Point2D, width : <b>int</b>, height : <b>int</b>)</li><li>+ getArea(): <b>int</b></li><li>+ containsPoint(other : Point2D): <b>boolean</b></li></ul>

Aber wie greifen wir jetzt auf die Attribute zu?

# Getter/Setter

- ▶ Zugriff auf Attribute über **Getter/Setter**

- ▶ **Getter** liefert Wert

```
public Typ getAttribut(){  
    return attribut;  
}
```

Kann von meisten IDEs **generiert werden**

- ▶ **Setter** setzt Wert

```
public void setAttribut(Typ attribut){  
    if (Attribut ungültig)  
        throw new IllegalArgumentException("Ungültig!");  
    this.attribut = attribut;  
}
```

Kann (bis auf Prüfung) auch von IDE **generiert werden**

- ▶ Nur **Getter**: „read-only“

- ▶ Prinzipiell kann jede Methode einer Klasse den **Objektzustand** ändern

## SimpleRectangle

– center : Point2D

– width : **int**

– height : **int**

+ SimpleRectangle(center : ↵  
Point2D, width : **int**, height : **int**)

+ getArea(): **int**

+ containsPoint(other : Point2D): **boolean**

+ getCenter(): Point2D

+ setCenter(center : Point2D)

+ getWidth(): **int**

+ setWidth(**int** width)

+ getHeight(): **int**

+ setHeight(**int** height)

# Vorteile von Gettern/Settern

- ▶ Erhalt der Objektkonsistenz
- ▶ Debugging von Zugriffen

```
public void getHeight(){  
    log("getHeight() aufgerufen");  
    return height;  
}
```

- ▶ Zugriff auf „virtuelle“ Attribute:

```
public int getArea(){  
    return width * height;  
}
```

- ▶ Performance: Hinauszögern von Update-Operationen

```
public int getX(){  
    if (stateChanged)  
        x = computeX();  
    return x;  
}
```



## private und Getter/Setter reichen nicht

- ▶ **private** bietet keinen Schutz vor „böswilligem“ Zugriff
  - ▶ Sichtbarkeit wird zur **Übersetzung** geprüft
  - ▶ **nicht** zur Laufzeit
  - ▶ „böswilliger“ Code kann über **Reflection-API** Daten ändern
  - ▶ Sichtbarkeit ist ein **Schnittstellen-Vertrag**
- ▶ **private** **schützt nicht** vor Zugriff von Objekten der **gleichen Klasse**
  - ▶ SimpleRectangle-Objekt **darf** auf **private/protected** Attribute anderer SimpleRectangle-Objekte **zugreifen**

```
public boolean isLargerThan(SimpleRectangle other){  
    other.width = 0; //muahaha  
    other.height = 0;  
    return true;  
}
```

- ▶ **Besser**: nicht machen („code smell“)
- ▶ **Sicherer**: unveränderliche Klassen (später)

# Inhalt

## Datenkapselung

Allgemeines Konzept: Schnittstellenvertrag

# Schnittstellenvertrag

- ▶ **Schnittstelle** einer Klasse:
  - ▶ „Alles was nicht **private** ist“
  - ▶ Definiert **was** eine Klasse anbietet
  - ▶ **Vertrag/Protokoll**: Vereinbarung zwischen Klasse und Nutzer
- ▶ **Designprinzip**
  - ▶ **Ockhams Rasiermesser**: So **klein** wie **möglich**, sie **groß** wie **nötig**
  - ▶ **Weil**: Alles was öffentlich sichtbar ist, schafft **Abhängigkeiten**
- ▶ Schnittstelle von SimpleRectangle

## SimpleRectangle

```
+ SimpleRectangle(center : Point2D, width : int, height : int)  
+ getArea(): int  
+ containsPoint(other : Point2D): boolean  
+ getCenter(): Point2D  
+ setCenter(center : Point2D)  
+ getWidth(): int  
+ setWidth(int width)  
+ getHeight(): int  
+ setHeight(int height)
```

# Implementierung

- ▶ Implementierung einer Klasse
  - ▶ Alles was **nicht Schnittstelle** ist
    - ▶ **private** Attribute
    - ▶ **private** Methoden (z.B. Hilfsmethoden)
    - ▶ Rümpfe der Methoden
    - ▶ ...
  - ▶ Definiert **wie** die Klasse ihre Funktion bereitstellt
- ▶ Geht **nur die Klasse** was an
- ▶ Implementierung von **SimpleRectangle**

## SimpleRectangle

- center : Point2D
- width : **int**
- height : **int**

```
public void getArea() { // Schnittstelle
    return width * height; // Implementierung
}
```

# Nicht-Programmier Beispiel: PC

## Desktop-PC

- ▶ Schnittstelle
  - ▶ **Eingabe:** Tastatur, Maus, Schalter
  - ▶ **Ausgabe:** Bildschirm (UI), LEDs, Lautsprecher
- ▶ Implementierung
  - ▶ **Software:** Applikationen, Betriebssystem
  - ▶ **Hardware:** CPU, RAM, Mainboard
  - ▶ **Physik:** Ströme, Elektronen, Quanteneffekte
- ▶ Eigentlich: **Hierarchie** von Schnittstellen
  - ▶ **Modularisierung**
  - ▶ Jedes Modul hat Schnittstelle (z.B. „Pins“ der CPU)
  - ▶ Nur so ist **Komplexität** beherrschbar



## Unveränderliche Klasse

Motivation

Das Schlüsselwort `final`

Initialisierung von `final` Objektvariablen

Definition: Unveränderliche Klasse

Arbeiten mit unveränderlichen Klassen

Warum überhaupt unveränderliche Klassen?

Beispiele

# Inhalt

## Unveränderliche Klasse

### Motivation

## Motivation: Böses isLargerThan

- ▶ Zur Erinnerung: Böse Methode aus SimpleRectangle

```
public boolean isLargerThan(SimpleRectangle other){  
    other.width = 0; //muahaha  
    other.height = 0;  
    return true;  
}
```

- ▶ **private** schützt **nicht** vor Schreibzugriff **innerhalb** der Klasse
- ▶ Manchmal nicht so **offensichtlich böse** wie oben

```
public void copyTo(SimpleRectangle other){  
    other.width = this.width;  
    other.height = this.height;  
}
```

- ▶ Ist das noch **OK**?
  - ▶ Allgemeine Meinung: **code smell**
- ▶ Wie kann man sich davor **schützen**?



# Inhalt

## Unveränderliche Klasse

Das Schlüsselwort final

# final

- ▶ **Modifizier:** Nur **eine** Zuweisung möglich, danach **unveränderlich**
- ▶ Wird nur zur **Übersetzungszeit** geprüft
- ▶ **Beispiele:**
  - ▶ **Variablendeklaration**

```
final int i = 0;  
i = 1; // FEHLER
```

- ▶ **Parameter**

```
public boolean isLargerThan(final SimpleRectangle other){  
    other = this; // FEHLER  
    // (leider) immer noch möglich, da width nicht final  
    other.width = 0;  
}
```

- ▶ **Objektvariablen**

```
private final int width;  
private final int height;
```

width und height können nur **einmal** (z.B. im Konstruktor) zugewiesen werden

# ImmutableSimpleRectangle

## ► Unveränderliche Version von SimpleRectangle

### ► Attribute

```
7 private final Point2D center;  
8 private final int width;  
9 private final int height;
```

📄 shapes/ImmutableSimpleRectangle.java

### ► Konstruktor

```
13 public ImmutableSimpleRectangle(Point2D center, int width, int height){  
14     this.center = center;  
15     this.width = width;  
16     this.height = height;  
17 }
```

📄 shapes/ImmutableSimpleRectangle.java

# Unveränderliche Version von SimpleRectangle

## ► Getter

```
21 public Point2D getCenter() {  
22     return center;  
23 }  
25 public int getWidth() {  
26     return width;  
27 }  
29 public int getHeight() {  
30     return height;  
31 }
```

shapes/ImmutableSimpleRectangle.java

## ► Aber: Wo sind die Setter?

- Die kann es nicht geben, da die Attribute **final** sind
- Aber wie verändert man dann ein Objekt? **Gar nicht!**

# Inhalt

## Unveränderliche Klasse

Initialisierung von `final` Objektvariablen

# Initialisierung von **final** Objektvariablen

- ▶ **final** Objektvariablen **müssen** einmal zugewiesen werden

- ▶ Konstruktor

```
private final int answer;  
public Answer(int answer){  
    this.answer = answer;  
}
```

- ▶ Verkettete Konstruktoren

```
private final int answer;  
public Answer(int answer){  
    this.answer = answer;  
}  
  
public Answer(){  
    this(42);  
}
```

# Initialisierung von final Objektvariablen

- ▶ **final** Objektvariablen **müssen** einmal zugewiesen werden
  - ▶ **Initialisierung** bei Deklaration

```
private final int answer = 42;
```

- ▶ Initializer

```
private final int answer;  
{  
    answer = 42;  
}
```

# Initialisierung von final Objektvariablen

## ► Nicht möglich

### ► Defaultwert übernehmen

```
private final int answer; // Defaultwert 0
public int getAnswer() {
    return answer; // FEHLER
}
```

### ► Initialisierung über Methoden

```
private final int answer;
public Answer(int answer){
    setAnswer(answer);
}
public void setAnswer(int answer) {
    this.answer = answer; // FEHLER
}
```



# Inhalt

## Unveränderliche Klasse

Definition: Unveränderliche Klasse

## Definition: Unveränderliche Klasse

- ▶ Eine Klasse heißt **unveränderlich** (auch **immutable**), wenn der **Zustand** eines Objekts der Klasse nach der Konstruktion **nicht verändert** werden kann
- ▶ **Eigenschaften** unveränderlicher Klassen
  - ▶ Attribute sind **final**
  - ▶ Keine **Setter** oder Methoden, die Zustand ändern
  - ▶ **Oft**: Klasse selbst ist **final**, d.h. keine Ableitung erlaubt (später)
- ▶ **Beispiele** aus dem JDK
  - ▶ ↗ **String**
  - ▶ ↗ **Integer**, ↗ **Double**, etc.

# Inhalt

## Unveränderliche Klasse

Arbeiten mit unveränderlichen Klassen

## Wie „verändert“ man unveränderliche Klassen?

- ▶ Zustand eines unveränderlichen Objekt ist **fest**
- ▶ **Keine Änderung** möglich (ImmutableSimpleRectangle)

```
public void enlarge(int deltaWidth, int deltaHeight){  
    this.width += deltaWidth; // FEHLER  
    this.height += deltaHeight; // FEHLER  
}
```

- ▶ **Lösung:** Neues Objekt erstellen

```
46 public ImmutableSimpleRectangle enlarge(  
47     int deltaWidth, int deltaHeight){  
48     return new ImmutableSimpleRectangle(  
49         this.center,  
50         this.width + deltaWidth,  
51         this.height + deltaHeight);  
52 }
```

📄 shapes/ImmutableSimpleRectangle.java

## Unveränderliche Klasse

Warum überhaupt unveränderliche Klassen?

# Vorteile

- ▶ Keine **unkontrollierte Änderung** des Zustands

```
public boolean isLargerThan(SimpleRectangle other){  
    other.width = 0; // FEHLER  
    other.height = 0;  
    return true;  
}
```


- ▶ **Thread-Sicherheit**:
  - ▶ Bei **simultanen Schreibzugriff** auf ein Objekt aus unterschiedlichen Threads können Probleme auftreten („**race conditions**“)
  - ▶ Ist ein Objekt unveränderlich, gibt es das Problem nicht
- ▶ Keine **Setter**
- ▶ **Nachteile**
  - ▶ Höherer **Speicherbedarf**
  - ▶ Mehr **Rechenzeit**
  - ▶ Geringfügig größerer **Implementierungsaufwand**


# Inhalt

## Unveränderliche Klasse Beispiele

## Beispiel: String

### ► String:


- „verändernde“ Methoden konstruieren **neuen**  String

```
7   runImmutableStringExample  
8  String quote = "the cake is a lie!";  
9  quote.toUpperCase();  
10 System.out.print(quote);
```

 ImmutableStringExamples.java

the cake is a lie!

### ► Richtig:

```
16  runImmutableStringExample2  
17 String quote = "the cake is a lie!";  
18 String upperCaseQuote = quote.toUpperCase();  
19 System.out.print(upperCaseQuote);
```

 ImmutableStringExamples.java

THE CAKE IS A LIE!



# Beispiel: Clojure

## ► Clojure

- **Funktionale** Programmiersprache
- basiert auf **Java**
- **LISP**-Syntax („list processor“)
- **alle** Datenstrukturen sind **unveränderlich**
- Sehr geeignet für Programme mit **mehreren Threads**

## ► Beispiel

```
(def l [2 3 4]) ; Vector mit drei Zahlen  
(cons 1 l)      ; fügt 1 vorne ein  
[1 2 3 4]  
(print l)       ; Ausgabe  
[2 3 4]
```

Vector bleibt **unverändert**



# Inhalt

## Klassenvariablen und -Methoden

- Der Modifier `static`

- Klassenattribute

- Klassenmethoden

- Anwendungsbeispiel: Singleton-Pattern

## Klassenvariablen und -Methoden

Der Modifier static

# Der Modifier **static**

- ▶ Der Modifier **static** definiert
  - ▶ **Klassenattribute** und **-methoden**
  - ▶ bzw. **statische** Attribute/Methoden
- ▶ Vergleich mit **Objektattributen/-methoden**

	Statisch	Nicht-Statisch
Attribut	genau einmal	pro Objekt
Methode	Kontext Klasse	Kontext Objekt ( <b>this</b> )

- ▶ Statische Attribute verwendet man
  - ▶ Alles was eher in den **Kontext der Klasse** als des Objekts passt
  - ▶ **Konstanten**
  - ▶ Alles was nur „**einmal existieren**“ darf
  - ▶ **Utility-Methoden** (z.B. ↗ **Math.cos()**)
  - ▶ **Methoden** die im Kontext der Klasse ausgeführt werden

# Inhalt

## Klassenvariablen und -Methoden Klassenattribute

# Klassenattribute

- ▶ **Attribute** einer Klasse mit dem **static**-Modifizier

```
public class GlobalCounter{  
    public static int value = 1;  
}
```

- ▶ Existieren im Kontext der **Klasse**
- ▶ ... nur **einmal**
- ▶ Zugriff:
  - ▶ **Innerhalb** der Klasse wie **Objektvariable** (ohne **this!**)

```
public void incCounter(){  
    value++;  
}
```

- ▶ **Außerhalb** der Klasse über **Klassenname**


```
print(GlobalCounter.value);
```

## Beispiel: ConfigurableGreeter

```
4 public class ConfigurableGreeter {  
6     public static String greeting = "Hello";  
8     private String target;  
10    public ConfigurableGreeter(String target){  
11        this.target = target;  
12    }  
14    public void greet(){  
15        System.out.printf("%s, %s!\n", greeting, target);  
16    }  
18 }
```

ConfigurableGreeter.java

## Beispiel: ConfigurableGreeter

```
8   runConfigurableGreeterExample  
9  ConfigurableGreeter landshutGreeter =  
10     new ConfigurableGreeter("Landshut");  
11  ConfigurableGreeter studentGreeter =  
12     new ConfigurableGreeter("Students");  
14  landshutGreeter.greet();  
15  studentGreeter.greet();  
17  ConfigurableGreeter.greeting = "Servus";  
18  landshutGreeter.greet();  
19  studentGreeter.greet();
```

 ConfigurableGreeterExample.java

```
Hello, Landshut!  
Hello, Students!  
Servus, Landshut!  
Servus, Students!
```



# Initialisierung

## ► Default-Wert

```
public static int value; // Default-Wert 0
```

## ► Bei der Deklaration

```
public static String greeting = "Hello";
```

## ► Statischer Initializer

```
public static String greeting;  
static{  
    greeting = "Hello";  
}
```

Wird beim **Laden der Klassendeklaration** ausgeführt

# Anwendungsbeispiel: Konstanten

## ► Konstanten

- **public** — Zugriff für jeden
- **static** — hängen nicht von Objekt ab
- **final** — bleiben im Wert gleich
- **Bezeichner**: SCREAMING\_SNAKE\_CASE

## ► Beispiele

- [↗](#) Math.PI

```
public static final double PI = 3.14159265358979323846;
```

- CelestialBody.GRAVITATIONAL\_CONSTANT

```
public static final double  
    GRAVITATIONAL_CONSTANT = 6.67430e-11;
```

- **Schlechtes Beispiel** (wurde so gemacht bevor es **enums** gab):

```
public static final int RED = 0;  
public static final int GREEN = 1;  
public static final int BLUE = 2;
```

## Anwendungsbeispiel: Seriennummer


Produkten soll eine **fortlaufende, eindeutige Seriennummer** gegeben werden

```
4 public class Product {  
5     private static int serialNumberCounter = 0;  
7     private final int serialNumber;  
8     private final String name;  
10    public Product(String name) {  
11        serialNumberCounter++;  
12        this.serialNumber = serialNumberCounter;  
13        this.name = name;  
14    }  
16    public int getSerialNumber() { return serialNumber; }  
17    public String getName() { return name; }  
18 }
```

Product.java

# Anwendungsbeispiel: Seriennummer

## ► Verwendung

```
7  runProductExample  
8 Product sword = new Product("Sword");  
9 Product shield = new Product("Shield");  
11 System.out.printf("%s (%d)%n",  
12     sword.getName(), sword.getSerialNumber());  
14 System.out.printf("%s (%d)%n",  
15     shield.getName(), shield.getSerialNumber());
```

 ProductExample.java

Sword (1)  
Shield (2)

## ► Statische Variable serialNumberCounter

- wird bei Erstellung eines Product-Objekts **erhöht**
- ist dadurch **fortlaufend** und **eindeutig**

# Lebensdauer einer Klassenvariable

- ▶ Lebensdauer einer Klassenvariable
  - ▶ Von: Klasse wird geladen/initialisiert
  - ▶ Bis: Programm wird beendet
- ▶ Probleme
  - ▶ ungültiger Wert bleibt (unter Umständen) bis zum Programmende erhalten
  - ▶ erschwert Fehlersuche
  - ▶ Prinzip ähnlich wie bei globalen Variablen (in C)
  - ▶ Zugriff kann nicht kontrolliert werden
- ▶ Daher:
  - ▶ Vorsichtig sein
  - ▶ Besser: nicht-statisch, **final** oder zumindest **private**

# Inhalt

## Klassenvariablen und -Methoden

### Klassenmethoden

# Klassenmethoden

- ▶ Methoden einer Klasse mit dem **static**-Modifizier

```
public class StaticGreeter{  
    public static void printGreeting(String target){  
        System.out.printf("Hello, %s%n", target);  
    }  
}
```

- ▶ Werden im Kontext der **Klasse** ausgeführt
- ▶ Können **nicht** auf Objektvariablen zugreifen
- ▶ Zugriff:
  - ▶ Innerhalb der Klasse wie **Methode** (ohne **this**!)

```
printGreeting("Landshut");
```

- ▶ Außerhalb der Klasse über **Klassenname**

```
StaticGreeter.printGreeting("Landshut");
```

# Zugriff innerhalb statischer Methoden

- ▶ Statische Methoden können auf **keine Objektvariablen/-methoden** zugreifen, nur auf **Klassenattribute/-methoden**

```
private static String greeting = "Hello";  
private String target = "World";  
public static void greet(){  
    System.print(greeting + ", "); // funktioniert  
    System.print(target + "!"); // FEHLER  
}
```

- ▶ Entsprechend existiert **this** im statischen Kontext nicht

```
public static void accessThis(){  
    this.var++; // FEHLER  
}
```



# Anwendungsbeispiel: Utility-Klassen

- ▶ Utility-Klassen sind eine Ansammlung von **statischen Hilfsmethoden**
- ▶ Die Klasse `Math`: Beinhaltet mathematische Hilfsmethoden

```
public class Math{  
    public static double abs(double a){ /* ... */ }  
    public static double sin(double a){ /* ... */ }  
    public static double cos(double a){ /* ... */ }  
    public static double max(int a, int b){ /* ... */ }  
    public static double round(int a, int b){ /* ... */ }  
    /* ... */  
}
```

# Anwendungsbeispiel: Zugriff auf statische Attribute

## ► Beispiel von vorher

```
4 public class ConfigurableGreeter {  
6     public static String greeting = "Hello";  
8     private String target;  
10    public ConfigurableGreeter(String target){  
11        this.target = target;  
12    }  
14    public void greet(){  
15        System.out.printf("%s, %s!\n", greeting, target);  
16    }  
18 }
```

ConfigurableGreeter.java

# Anwendungsbeispiel: Zugriff auf statische Attribute

## ► Problematisch:

```
public static String greeting = "Hello";
```

## ► Verletzt Prinzip der Datenkapselung

```
ConfigurableGreeter.greeting = null; // muahaha
```

## ► Besser

```
private static String greeting = "Hello";  
public static void setGreeting(String newGreeting){  
    if (newGreeting == null)  
        throw new IllegalArgumentException("...");  
    greeting = newGreeting  
}  
public static String getGreeting() { return greeting; }
```

# Inhalt

## Klassenvariablen und -Methoden

Anwendungsbeispiel: Singleton-Pattern

# Singleton-Pattern


- ▶ Singleton-Pattern
  - ▶ Problem:
    - ▶ Von einer Klasse soll es **höchstens ein** Objekt geben
    - ▶ Soll erst **erstellt** werden, wenn es **gebraucht** wird (Ressourcen schonen)
  - ▶ Lösungsansatz: Singleton-Pattern
  - ▶ Design-Pattern der objektorientierten Programmierung (mehr in „Software Engineering“)
- ▶ So funktioniert es:
  - ▶ Nur Klasse selbst darf Objekt **erstellen**: **private** Konstruktor
  - ▶ Es darf **nur ein** Objekt geben: **Klassenattribut** hält Referenz
  - ▶ Erstellung bei **erstem Zugriff**: statische Methode getInstance
    - ▶ **erstellt** Objekt wenn noch nicht existent
    - ▶ **sonst**: gibt Referenz zurück

## SingletonGreeter I

```
4 public class SingletonGreeter {  
5     private static SingletonGreeter instance;  
7     private String greeting;  
9     private SingletonGreeter(){  
10         System.out.println("SingletonGreeter()");  
11         greeting = "Hello";  
12     }  
14     public static SingletonGreeter getInstance(){  
15         System.out.println("getInstance()");  
17         if (instance == null)  
18             instance = new SingletonGreeter();  
19         return instance;  
20     }  
22     public void greet(String target){  
23         System.out.printf("%s, %s!\n", greeting, target);  
24     }  
25 }
```

- ▶ Konstruktor — **private**
- ▶ **instance** — statisch, hält (einzige) Referenz
- ▶ **getInstance** — statisch, **erstellt Instanz**, wenn nötig, und **liefert** diese **zurück**

## SingletonGreeter

```
7   runSingletonGreeterExample
8  System.out.println("Los geht's!");
10  var greeter = SingletonGreeter.getInstance();
11  greeter.greet("Landshut");
13  var greeter2 = SingletonGreeter.getInstance();
14  greeter2.greet("Students");
```

 SingletonGreeterExample.java

```
Los geht's!
getInstance()
SingletonGreeter()
Hello, Landshut!
getInstance()
Hello, Students!
```

- ▶ Konstruktor wird nur **einmal** aufgerufen
- ▶ ...und erst dann wenn **getInstance** aufgerufen wird



# Inhalt

## Enumerationen

Motivation

Enumerationen: Grundversion

Enumerationen: Vollständige Version

Hilfsmethoden


Wann Enumerationen verwenden?

# Inhalt

## Enumerationen

### Motivation

# Wochentage

- ▶ Wir wollen **Wochentage** modellieren
- ▶ Funktionen
  - ▶ **Konvertierung** in deutschen Namen
  - ▶ **Abfrage**: Werktag?
- ▶ **Alphaversion**: Klasse WeekdayAlpha  WeekdayAlpha.java
- ▶ **Idee**: Werkstage als Konstanten

```
6 public static final int MONDAY = 0;  
7 public static final int TUESDAY = 1;  
8 public static final int WEDNESDAY = 2;  
9 public static final int THURSDAY = 3;  
10 public static final int FRIDAY = 4;  
11 public static final int SATURDAY = 5;  
12 public static final int SUNDAY = 6;
```

 WeekdayAlpha.java

# Wochentage: 1. Version

## ► getGermanName

```
16 public static String getGermanName(int weekday){
17     switch (weekday) {
18         case MONDAY: return "Montag";
19         case TUESDAY: return "Dienstag";
20         case WEDNESDAY: return "Mittwoch";
21         case THURSDAY: return "Donnerstag";
22         case FRIDAY: return "Freitag";
23         case SATURDAY: return "Samstag";
24         case SUNDAY: return "Sonntag";
25         default:
26             throw new IllegalArgumentException("Invalid weekday");
27     }
28 }
```

WeekdayAlpha.java

## ▶ isWorkday

```
32 public static boolean isWorkday(int weekday){  
33     if (weekday < 0 || weekday > 6)  
34         throw new IllegalArgumentException("Invalid weekday");  
35     return weekday < 5;  
36 }
```

WeekdayAlpha.java

## ▶ Unschön

- ▶ Nicht **Typsicher**: **int** kann beliebigen Wert annehmen
- ▶ Fallunterscheidungen: langer **switch-case**

## ▶ Besser: **enum**

# Inhalt

## Enumerationen

### Enumerationen: Grundversion

# Enumerationen: Grundversion

```
public enum EnumIdentifier { WERT_1, WERT_2, ..., WERT_N }
```

## ► Definition

- **public** — Sichtbarkeit (wie bei Klassen)
- **enum** — Schlüsselwort
- **EnumIdentifier** — Name der Enumeration
- **WERT\_i** — Werte der Enumeration („screaming snake case“)

## ► Deklaration in Datei mit **enum**-Namen (z.B. EnumIdentifier.java)

## ► Oder: innerhalb einer Klasse (**später**)

## ► Wochentage — final!

```
4 public enum WeekdayBeta {  
5     MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY  
6 }
```

WeekdayBeta.java

## ► Zugriff über **enum**-Bezeichner WeekdayBeta.MONDAY

## WeekdayBetaUtils

- ▶ getGermanName und isWorkday sind nun in **separater** Klasse WeekdayBetaUtils
- ▶ getGermanName

```
6 public static String getGermanName(WeekdayBeta weekday){
7     switch (weekday) {
8         case MONDAY: return "Montag";
9         case TUESDAY: return "Dienstag";
10        case WEDNESDAY: return "Mittwoch";
11        case THURSDAY: return "Donnerstag";
12        case FRIDAY: return "Freitag";
13        case SATURDAY: return "Samstag";
14        case SUNDAY: return "Sonntag";
15        // never happens (or will it?)
16        default: return null;
17    }
18 }
19 }
```

WeekdayBetaUtils.java



▶ isWorkday

```
23 public static boolean isWorkday(WeekdayBeta weekday){  
24     return (weekday != WeekdayBeta.SATURDAY &&  
25         weekday != WeekdayBeta.SUNDAY);  
26 }
```

WeekdayBetaUtils.java

▶ Schöner: Typsicher!

▶ Unschön:

- ▶ Immer noch Fallunterscheidungen
- ▶ Funktion von Datendeklaration getrennt: WeekdayBeta, WeekdayBetaUtils

# Wochentage — final 2! I

## Deklaration des `enums` innerhalb der Klasse

```
4 public class WeekdayGamma
5 {
6     public enum Weekday {
7         MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
8     }
10    public static String getGermanName(Weekday weekday){
11        switch (weekday) {
12            case MONDAY: return "Montag";
13            case TUESDAY: return "Dienstag";
14            case WEDNESDAY: return "Mittwoch";
15            case THURSDAY: return "Donnerstag";
16            case FRIDAY: return "Freitag";
17            case SATURDAY: return "Samstag";
18            case SUNDAY: return "Sonntag";
19            // never happens (or will it?)
20            default: return null;
21        }
22    }
```

## Wochentage — final 2! II

```
24 public static boolean isWorkday(Weekday weekday){  
25     return (weekday != Weekday.SATURDAY &&  
26         weekday != Weekday.SUNDAY);  
27 }  
28 }
```

WeekdayGamma.java

- ▶ **Schöner:** Funktion und Datendeklaration an einer Stelle
- ▶ **Unschön**
  - ▶ Immer noch **Fallunterscheidungen**
  - ▶ Zugriff von außen **umständlicher**: WeekdayGamma.Weekday.MONDAY

# Inhalt

## Enumerationen

Enumerationen: Vollständige Version

# Enumeration: Vollständige Version

- ▶ **enum** erlaubt **Definition** von
  - ▶ **Attributen**
  - ▶ (privaten) **Konstruktoren**
  - ▶ **Methoden**
- ▶ **Beispiel:** Gewichtsmaße

```
5 public enum WeightUnit
6 {
7     GRAM("g"), KILOGRAM("kg"), TON("t"), POUND("lb");
8
9     private final String symbol;
10
11     WeightUnit(String symbol) {
12         this.symbol = symbol;
13     }
14
15     public String getSymbol() { return symbol; }
16 }
```

WeightUnit.java

# Enumeration: Vollständige Version

- ▶ **enum** sind **spezielle Klassen** in Java
- ▶ **Deklaration**: Sichtbarkeit und Bezeichner

```
public enum WeightUnit
```

- ▶ **Werte**:

```
GRAM("g"), KILOGRAM("kg"), TON("t"), POUND("lb");
```

- ▶ **Konstruktoraufrufe**
  - ▶ Definiert die **einigen** Instanzen der Enumeration
  - ▶ Mindestens eine
- ▶ **Attribute**

```
private final String symbol;
```

- ▶ Wie in **Klassendefinition**
  - ▶ D.h. auch **static** und **andere Modifier** erlaubt

# Enumeration: Vollständige Version

## ► Konstruktor

```
private WeightUnit(String symbol) {  
    this.symbol = symbol;  
}
```

- Muss **private** sein

## ► Methoden

```
public String getSymbol() {  
    return symbol;  
}
```

- Wie in **Klassen**
- D.h. auch **static** und **andere Modifier** erlaubt

## Enumeration: Unter der Haube

`WeightUnit` wird übersetzt in (gekürzt, vgl. `javap -p`-Ausgabe)

```
public final class WeightUnit extends Enum<WeightUnit> {  
    public static final WeightUnit GRAM = new WeightUnit("g");  
    public static final WeightUnit KILOGRAM =  
        new WeightUnit("kg");  
    public static final WeightUnit TON = new WeightUnit("t");  
    public static final WeightUnit POUND = new WeightUnit("lb");  
    private String symbol;  
    private WeightUnit(String symbol){  
        this.symbol = symbol;  
    }  
    public String getSymbol(){ return symbol; }  
}
```



# Wochentage — final 3! I

## ► Wochentage — final 3!

```
4 public enum Weekday {
5     MONDAY("Montag", true),
6     TUESDAY("Dienstag", true),
7     WEDNESDAY("Mittwoch", true),
8     THURSDAY("Donnerstag", true),
9     FRIDAY("Freitag", true),
10    SATURDAY("Samstag", false),
11    SUNDAY("Sonntag", false);
12
13    private final boolean isWorkday;
14    private final String germanName;
15
16    private Weekday(String germanName, boolean isWorkday){
17        this.germanName = germanName;
18        this.isWorkday = isWorkday;
19    }
20
21    public boolean isWorkday() {
```

## Wochentage — final 3! II


```
24     return isWorkday;  
25 }  
27 public String getGermanName(){  
28     return germanName;  
29 }  
30 }
```

Weekday.java

- ▶ Schön
  - ▶ Kompakte Definition
  - ▶ Typsicher
  - ▶ Keine Fallunterscheidungen mehr
- ▶ Unschön (aber trotzdem schönste Lösung):
  - ▶ Mehr Speicherbedarf als in der ersten Version

# Wochentage: Verwendung

## ► Beispiel für Verwendung

```
6   runWeekdayExample
7  public static void weekdayExample(Weekday day) {
8      System.out.printf("%s: ", day.getGermanName());
10     if (day.isWorkday()){
11         if (day == Weekday.FRIDAY){
12             System.out.println("Hoch die Hände, Wochenende!");
13         }else{
14             System.out.println("An die Arbeit!");
15         }
16     }else{
17         System.out.println("Yes, weekend!");
18     }
19 }
```

 WeekdayExamples.java

# Wochentage: Verwendung

## ► Ausgabe

```
Montag: An die Arbeit!  
Dienstag: An die Arbeit!  
Mittwoch: An die Arbeit!  
Donnerstag: An die Arbeit!  
Freitag: Hoch die Hände, Wochenende!  
Samstag: Yes, weekend!  
Sonntag: Yes, weekend!
```

## ► Zugriff erfolgt wie auf **statische, konstante Attribute**

```
Weekday.MONDAY
```

## ► Identität über ==


```
day == Weekday.MONDAY
```

## ► Zugriff auf Methoden/Attribute wie bei Objekten

```
Weekday.MONDAY.isWorkday()
```

# Enumerationen und switch-case

Bei **switch-case** ist der Bezeichner des **enums** **nicht** nötig

```
25  runEnumSwitchCaseExample
26 switch (day){
27     case MONDAY: case TUESDAY: case WEDNESDAY:
28         System.out.println("Hmpff!");
29         break;
31     case THURSDAY: case FRIDAY:
32         System.out.println("Eigentlich schon Wochenende!");
33         break;
35     case SATURDAY: case SUNDAY:
36         System.out.println("Wochenende!");
37         break;
38 }
```


 WeekdayExamples.java

# Inhalt

## Enumerationen Hilfsmethoden

## Hilfsmethoden der Klasse `Enum<T>`

- ▶ Jedes `enum` leitet von Klasse `Enum<T>` ab
- ▶ Erbt nützliche **Hilfsmethoden**
  - ▶ **`static T[] values()`** — alle Werte des `enums` als Array

```
44  runEnumValuesExample  
45 for (var day : Weekday.values())  
46     System.out.printf("%s\n", day.getGermanName());
```

 WeekdayExamples.java

- ▶ **`static T valueOf(String s)`** — gibt `enum`-Wert zu `String`

```
52 Weekday.valueOf("FRIDAY"); // == Weekday.FRIDAY;
```

 WeekdayExamples.java


- ▶ **`String name()`** — liefert Namen

```
Weekday.WEDNESDAY.name() == "WEDNESDAY";
```

## Hilfsmethoden der Klasse `Enum<T>`

### ► Hilfsmethoden

- `int ordinal()` — liefert Ordnungszahl (vgl. `WeekdayAlpha`)

```
58  runEnumOrdinalExample  
59 for (var day : Weekday.values())  
60     System.out.printf("%s: %d\n", day.name(), day.ordinal());
```

 WeekdayExamples.java

```
MONDAY: 0  
TUESDAY: 1  
WEDNESDAY: 2  
THURSDAY: 3  
FRIDAY: 4  
SATURDAY: 5  
SUNDAY: 6
```

- **Hinweis:** Der Wert einer `enum`-Variable kann auch `null` sein

```
Weekday noDay = null;
```



# Inhalt

## Enumerationen

Wann Enumerationen verwenden?

# Enumerationen: Wann verwenden?

- ▶ Enumerationen für **endliche Wertemengen**
  - ▶ **Aufzählungen**: Wochentage, Monate, Grundfarben
  - ▶ **Zustände**: z.B. Kaffeeautomat, IDLE, BREWING, CLEANING
  - ▶ **Optionen, Operationen**: z.B. SQL-Kommandos SELECT, UPDATE, DELETE
- ▶ Semantik anderer Typen **nicht** „verbiegen“
  - ▶ **Statt**: **boolean** isFemale
  - ▶ **Besser**: **enum** { MALE, FEMALE, DIVERSE }
  - ▶ **Statt**:

```
void execute(String command){  
    if (command.equals("UPDATE"))  
        /* ... */  
    else if (command.equals("DELETE"))  
        /* ... */  
}
```

- ▶ **Besser**:

```
enum Command { UPDATE, DELETE, ... }  
void execute(Command command){ }
```

# Inhalt

## Kopieren

Kopieren über Wertzuweisung

Tiefe Kopie

Ergänzungen

# Inhalt

## Kopieren

Kopieren über Wertzuweisung

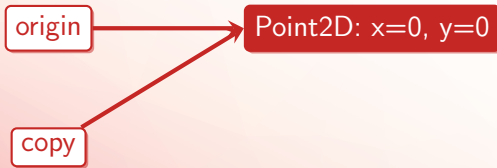
# Wertzuweisung

- ▶ **Primitive Typen:** Wert wird kopiert

```
double pi = 3.1415;  
double copy = pi;
```

- ▶ **Referenztypen:**

```
Point2D origin = new Point2D(0,0);  
Point2D copy = origin;
```



- ▶ Referenz wird kopiert
- ▶ Beide Referenzen zeigen auf **selbes Objekt**
- ▶ Wie erstellt man ein **Duplikat** eines Objekts?

# Flache Kopien

- ▶ Point2D hat Kopier-Konstruktor

```
24 public Point2D(Point2D other){  
25     this(other.getX(), other.getY());  
26 }
```

shapes/Point2D.java

- ▶ Kopiert Werte für x und y von anderem Objekt
- ▶ Beispiel

```
Point2D origin = new Point2D(0,0);  
Point2D copy = new Point2D(origin);
```



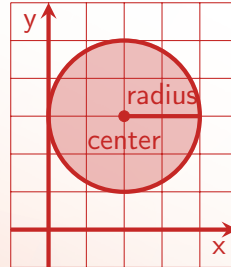
- ▶ Attribute des Objekts werden **kopiert**
- ▶ Beide Referenzen zeigen auf **unterschiedliche** Instanzen
- ▶ ...die aber in den **Werten** gleich sind

# Flache Kopien

- ▶ Point2D besitzt zwei **primitive Attribute** `int x`, `int y`
- ▶ Was passiert bei **Referenzen**?
- ▶ Die Klasse `Circle`

## Circle

```
- center : Point2D  
- radius : int  
+ Circle(center : Point2D, radius : int)  
+ Circle(other : Circle)  
+ getRadius(): int  
+ setRadius(radius : int)  
...
```



# Flache Kopie

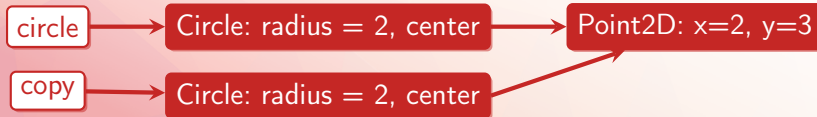
- ▶ Circle hat einen **Kopier-Konstruktor**

```
18 public Circle(Circle other){  
19     this.center = other.getCenter();  
20     this.radius = other.getRadius();  
21 }
```

shapes/Circle.java

- ▶ Beispiel

```
Point2D point = new Point2D(2,3);  
Circle circle = new Circle(point, 2);  
Circle copy = new Circle(circle);
```



- ▶ circle und copy zeigen auf dasselbe Point2D-Objekt



# Flache Kopien

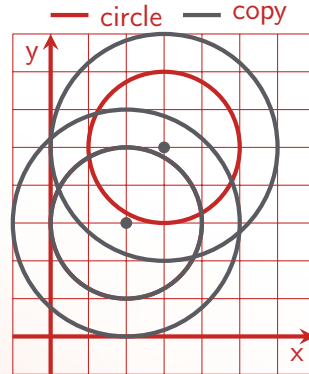
- ▶ Radius der **Kopie** ändern

```
copy.setRadius(3);
```

- ▶ **Keine Auswirkung** auf Original
- ▶ Zentrum der Kopie **verschieben**

```
copy.getCenter().move(1,2);
```

- ▶ Verschiebt **beide** Kreise
- ▶ **Grund**: Beide haben Referenz auf dasselbe Point2D-Objekt
  - ▶ **Flache Kopien**: Attribute werden mit **Wertzuweisung** kopiert
  - ▶ **Auswirkungen**
    - ▶ **Primitive Typen**: keine
    - ▶ **Referenztypen**: dahinterliegende Instanzen bleiben **dieselben**



# Inhalt

## Kopieren

Tiefe Kopie

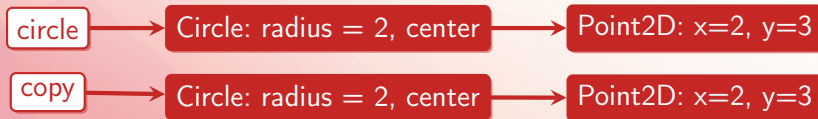
# Tiefe Kopie

- ▶ Tiefe Kopie: Alternativer **Kopier-Konstruktor** in Circle

```
public Circle(Circle other){  
    this.center = new Point2D(other.getCenter());  
    this.radius = other.getRadius();  
}
```

- ▶ Unterschied zu **flacher Kopie**: center wird **kopiert**
- ▶ Beispiel

```
Point2D point = new Point2D(2,3);  
Circle circle = new Circle(point, 2);  
Circle copy = new Circle(circle);
```



- ▶ circle und copy zeigen auf **unterschiedliche** Point2D-Objekt

# Tiefe Kopie

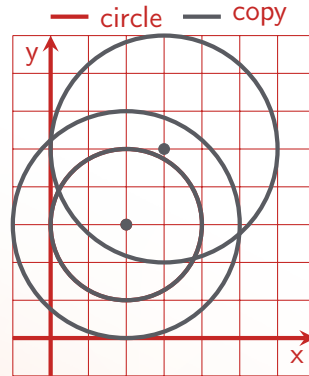
- ▶ Radius der **Kopie** ändern

```
copy.setRadius(3);
```

- ▶ **Keine Auswirkung** auf Original
- ▶ Zentrum der Kopie **verschieben**

```
copy.getCenter().move(1,2);
```

- ▶ Verschiebt **nur** Kopie
- ▶ **Grund**: Jedes Objekt hat sein **eigenes** Zentrum
  - ▶ **Tiefe Kopien**: Attribute werden **tief** kopiert
  - ▶ **Auswirkungen**
    - ▶ **Primitive Typen**: über Wertzuweisung
    - ▶ **Referenztypen**: (rekursiver) Kopiervorgang



# Inhalt

Kopieren

Ergänzungen

# Ergänzungen

- ▶ Bei einer tiefen Kopie
  - ▶ Aufrufer muss sich darauf verlassen, dass referenzierte Objekte **auch** tiefe Kopien erstellen

```
public Deeper(Deeper other){  
    this.deep = new Deep(other.getDeep());  
}
```

- ▶ Was ist wenn Kopier-Konstruktor von Deep **nicht tief** kopiert?
  - ▶ Dann ist gesamte Kopie **nicht tief**
- ▶ Aufpassen bei **Vererbung** (später)
  - ▶ Unterklassen müssen ebenfalls **Kopier-Konstruktor** „richtig“ implementieren
- ▶ **Später**: Weiterer Mechanismus zum Kopieren

```
Circle copy = (Circle) circle.clone();
```

# Inhalt

## Identität und Gleichheit

Identität

Gleichheit

# Inhalt

## Identität und Gleichheit

### Identität




# Identität

- ▶ Zwei Referenzen sind **identisch**, wenn sie auf **dasselbe** Objekt zeigen



- ▶ Identität entspricht **physischer Gleichheit** (gleiche Speicheradresse)
- ▶ Es gilt: **Identität impliziert Gleichheit**
- ▶ Aber aus **Gleichheit** folgt nicht immer **Identität**!

## Gleich aber nicht identisch

```
13  runBadIdentityExample  
14 Scanner scanner = new Scanner(System.in);  
16 final String password = "1234";  
18 System.out.println("Enter Password");  
19 String input = scanner.next();  
21 if (password == input)  
22     System.out.println("Access Granted!");  
23 else  
24     System.out.println(  
25         "Ah ah ah, you didn't say the magic word!");
```

 IdentityExamples.java

# Gleich aber nicht identisch

Enter Password

1234

Ah ah ah, you didn't say the magic word!

## ► Was ging hier schief?

- `scanner.next()` liest von Eingabe
- ...und erzeugt **neuen** String mit **Inhalt** „1234“



## ► Eingegebenes Passwort

- ist nicht **das selbe** wie das gespeicherte Passwort (**Identität**)
  - ist **das gleiche** wie wie das gespeicherte Passwort (**Gleichheit**)
- Wir müssen **Gleichheit** prüfen!


# Inhalt

## Identität und Gleichheit

### Gleichheit

# Gleichheit

- ▶ Was **heißt** Gleichheit zweier Objekte?
  - ▶ Zwei Objekte heißen **gleich**, wenn sie sich in jeder Hinsicht **nach außen hin gleich** Verhalten
  - ▶ Verhalten von Objekten wird durch (meistens alle) **Attribute** bestimmt
- ▶ **Wertgleichheit**
  - ▶ Zwei Objekte sind **wertgleich** wenn alle Ihre Attribute **wertgleich** sind
  - ▶ (**Wertgleichheit** impliziert **Gleichheit**)
- ▶ (**Wert-**)**Gleichheit** prüft man mit der Methode **equals**

```
39  runFixedIdentityExample  
40 if (password.equals(input))
```

 IdentityExamples.java

```
Enter Password  
1234  
Access Granted!
```

# Gleichheit bei eigenen Klassen

- ▶ equals wird von den Klassen des JDK implementiert
- ▶ Allgemeine Eigenschaften von equals
  - ▶ Äquivalenzrelation
    - ▶ Reflexiv: `x.equals(x)`
    - ▶ Symmetrisch: `x.equals(y)  $\iff$  y.equals(x)`
    - ▶ Transitiv: `x.equals(y)  $\wedge$  y.equals(z)  $\implies$  x.equals(z)`
  - ▶ null ist verschieden zu allem: `x.equals(null) == false`
  - ▶ Konsistenz: mehrfacher Aufruf von equals liefert immer das gleiche Ergebnis (vorausgesetzt Objekte werden nicht verändert)
- ▶ Wie implementiert man equals in eigenen Klassen?

# equals der Klasse Rectangle

- ▶ equals für die Klasse **Rectangle**

Rectangle
<ul style="list-style-type: none"><li>– center : Point2D</li><li>– width : <b>int</b></li><li>– height : <b>int</b></li></ul>
<ul style="list-style-type: none"><li>+ equals(other : Object): <b>boolean</b></li><li>...</li></ul>

- ▶ Zwei Rectangle-Objekte sind **gleich** wenn
  - ▶ sie gleiche **gleiche Breite und Höhe** haben
  - ▶ Ihre Mittelpunkte **gleich** sind

# equals — ein Kochrezept

## Ein Kochrezept

- ▶ **Signatur** erstellen

```
@Override public boolean equals(Object other)
```

- ▶ **Identität** prüfen

```
if (this == other)  
    return true;
```

Identität **impliziert** Gleichheit (Vergleich mit == geht **sehr schnell**)

- ▶ Auf **null** prüfen

```
if (other == null)  
    return false;
```

**null** **gleich** keinem Objekt



# equals — ein Kochrezept

## Ein Kochrezept

- ▶ Prüfe Gleichheit der Typen

```
if (getClass() != other.getClass())  
    return false;
```

**Achtung:** `instanceof` geht hier **nicht**, da abgeleitete Klassen sich **anders verhalten können** (später)

- ▶ Bisher Typ `Object`, jetzt Typ `Rectangle`

```
Rectangle otherRectangle = (Rectangle) other;
```

- ▶ Wertgleichheit der Attribute

- ▶ Höhe und Breite

```
if (height != otherRectangle.getHeight())  
    return false;  
if (width != otherRectangle.getWidth())  
    return false;
```

## equals — ein Kochrezept

- ▶ Wertgleichheit der **Attribute**
  - ▶ Mittelpunkt

```
if (!center.equals(otherRectangle.getCenter()))  
    return false;
```

- ▶ **Achtung:** Was ist wenn center **null** ist? ↗ `NullPointerException`
- ▶ **Verbesserte Version**

```
if (center == null) {  
    if (otherRect.getCenter() != null) return false;  
} else if (!center.equals(otherRectangle.getCenter()))  
    return false;
```

- ▶ ✗ sehr lange und immer der gleiche Code
- ▶ **Hilfsmethode**

```
if (!Objects.equals(center, otherRectangle.getCenter()))  
    return false;
```

- ▶ Zum **Schluss**, alle Tests bestanden: **return true;**

## equals der Klasse Rectangle I

```
89 @Override
90 public boolean equals(Object other) {
91     // Identitaet
92     if (this == other)
93         return true;
94
95     // null
96     if (other == null)
97         return false;
98
99     // Typvergleich
100    if (getClass() != other.getClass())
101        return false;
102
103    // Rectangle-cast
104    Rectangle otherRectangle = (Rectangle) other;
105    // Attribute vergleichen
106    if (height != otherRectangle.getHeight())
107        return false;
```

## equals der Klasse Rectangle II


```
109     if (width != otherRectangle.getWidth())
110         return false;
112     if (!Objects.equals(center, otherRectangle.getCenter()))
113         return false;
115     // Objekte sind gleich
116     return true;
117 }
```

shapes/Rectangle.java

## equals — ein Kochrezept

1. **Signatur:** `@Override public boolean equals(Object other)`
2. **Identiät prüfen:** `this == other`
3. **null prüfen:** `other == null`
4. **Typ prüfen:** `getClass() != other.getClass()`
5. **Cast:** z.B. `Rectangle otherRectangle = (Rectangle)other;`
6. **Attribute:** auf Wertgleichheit prüfen
  - ▶ **Primitive Typen:** direkter Vergleich mit `!=`
  - ▶ **Referenztypen:** `Objects.equals(x,y)`
7. **Alle Tests bestanden:** `return true;`

## Test von Rectangle.equals

```
51  runRectangleEqualsTest
52 Point2D p = new Point2D(2,3);
53 Rectangle rect1 = new Rectangle(p, 1, 2);
54 Point2D p2 = new Point2D(2,3);
55 Rectangle rect2 = new Rectangle(p2, 1, 2);
56
57 System.out.printf("rect1.equals(rect2): %b%n", rect1.equals(rect2));
58 System.out.printf("rect2.equals(rect1): %b%n", rect2.equals(rect1));
59 System.out.printf("rect1.equals(rect1): %b%n", rect1.equals(rect1));
60 System.out.printf("rect1.equals(null): %b%n", rect1.equals(null));
61 System.out.printf("rect1.equals(p): %b%n", rect1.equals(p));
62
63 rect2.setWidth(2);
64 System.out.printf("rect1.equals(rect2): %b%n", rect1.equals(rect2));
65 rect2.setWidth(1);
66
67 rect2.getCenter().move(1,1);
68 System.out.printf("rect1.equals(rect2): %b%n", rect1.equals(rect2));
```

 IdentityExamples.java

## Test von Rectangle.equals

```
rect1.equals(rect2): false ???  
rect2.equals(rect1): false ???  
rect1.equals(rect1): true  
rect1.equals(null): false  
rect1.equals(p): false  
rect1.equals(rect2): false  
rect1.equals(rect2): false
```

### ► Hier stimmt was nicht

- rect1.equals(rect2) und rect2.equals(rect1) sollten **true** liefern
- Genauere Untersuchung ergibt
  - Vergleich der Mittelpunkte, p.equals(p2), liefert **false**
  - Point2D implementiert equals **nicht**
  - Standard-Implementierung prüft nur **Identität**!

## Test von Rectangle.equals: 2. Versuch

- ▶ Nach Implementierungen von Point2D.equals

```
rect1.equals(rect2): true  
rect2.equals(rect1): true  
rect1.equals(rect1): true  
rect1.equals(null): false  
rect1.equals(p): false  
rect1.equals(rect2): false  
rect1.equals(rect2): false
```

- ▶ Jetzt passt's!
- ▶ Erkenntnis: equals nur dann korrekt wenn equals von referenzierten Klassen korrekt
- ▶ Ähnliche Situation wie bei tiefer Kopie



# Ergänzungen

- ▶ IDEs (z.B. Eclipse) unterstützen **automatische Generierung** von equals (**boilerplate code**)
- ▶ **Trotzdem**: Sie sollten wissen wie man equals implementiert (**Übung**)
- ▶ Java verlangt mit equals auch Implementierung von **hashCode**
  - ▶ Liefert **Hashwert** eines Objekts
  - ▶ Für **Einsortieren** in [HashMap](#) und Co.
  - ▶ **Schnelle Prüfung** von **Ungleichheit**

```
if (o1.hashCode() != o2.hashCode())  
    // Objekte können nicht gleich sein
```

- ▶ hashCode kann auch von IDE **generiert** werden

# Inhalt

## Dokumentation mit javadoc

Dokumentation: Psychologische Faktoren

JavaDoc — Inline Dokumentation

Erstellen der Dokumentation

Ergänzungen

# Inhalt

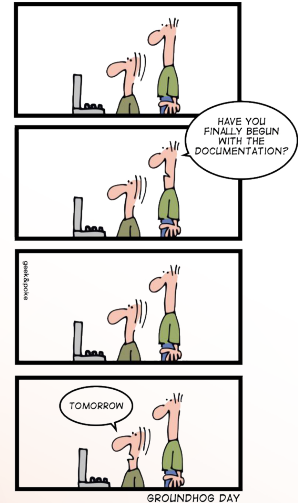
## Dokumentation mit javadoc

Dokumentation: Psychologische Faktoren

# Dokumentation: Psychologische und organisatorische Faktoren

- ▶ **Achtung:** Subjektive Beobachtungen folgen!
- ▶ **Entwickler**
  - ▶ schreiben (meist) **ungern** Dokumentation
  - ▶ verlassen **ungern** Ihre Entwicklungsumgebung
  - ▶ **schieben** Dokumentation gerne hinaus („Es ändert sich ja bestimmt noch was“)
  - ▶ **Schließlich:** Keine Zeit/kein Budget für Dokumentation
- ▶ **Extern** erstellte Dokumentation
  - ▶ Wikis wie **Confluence**, Word, etc.
  - ▶ Wird **nicht** gepflegt
  - ▶ Ist oft **uneinheitlich**

SIMPLY EXPLAINED



# Inhalt

## Dokumentation mit javadoc

JavaDoc — Inline Dokumentation

- ▶ JavaDoc: Dokumentation geschieht **direkt im Quellcode**

- ▶ Erstes Beispiel

```
99  
100  /**  
101   * Returns the distance between this and the other point.  
102   * The distance is Euclidean.  
103   *  
104   * @param other other point (must not be {@code null})  
105   * @return return Euclidean distance between the two points.  
106   */  
107  public double distance(final Point2D other){  
108      double dx = x - other.getX();  
109      double dy = y - other.getY();  
110      return Math.sqrt(dx*dx + dy*dy);  
111  }
```

shapes/Point2D.java

- ▶ javadoc **generiert** Dokumentation (z.B. HTML)

# JavaDoc — was kann dokumentiert werden?

- ▶ JavaDoc-Dokumentation für

- ▶ Klassen

```
/**  
 * This models a customer ...  
 */  
public class Customer{ /* ... */ }
```

- ▶ Methoden (s. oben)

- ▶ Objektvariablen

```
/**  
 * Ratio between a circle's circumference and diameter.  
 */  
public static final double PI = 3.14159265358979323846;
```

- ▶ Interfaces (später)

- ▶ Enumerationen

- ▶ Keine Dokumentation innerhalb von Methoden

# Aufbau von JavaDoc-Kommentaren

- ▶ **Anfang:** Blockkommentar mit zwei Sternen

```
/**
```

- ▶ **Zusammenfassung** (erster Satz mit Punkt abgeschlossen)

```
* Returns the distance between this and the other point.
```

- ▶ **Weitere Beschreibung** (optional)

```
* The distance is Euclidean.
```

- ▶ Zu dokumentierten Objekt **spezifische JavaDoc-Tags**

```
* @param other other point (must not be {@code null})  
* @return return Euclidean distance between the two points.
```

- ▶ **Abschluss**

```
*/
```



# JavaDoc-Tags

- ▶ JavaDoc-Tags sind durch @ markiert
- ▶ Klassen, **enums**, **Interfaces**
  - ▶ **@author**: Autor der Klasse

```
@author Handsome Jack
```

Hinweis: sollte (nicht mehr) verwendet werden

- ▶ **@version**: Version der Datei oder Releases

```
@version 1.0
```

Hinweis: sollte **automatisiert** befüllt werden (z.B. durch git)

- ▶ **Objektvariablen** haben keine eigenen Tags

# JavaDoc-Tags

## ► Methoden

```
/**  
 * Returns the maximum of two integer numbers.  
 * @param x first argument to max  
 * @param y second argument to max  
 * @return The larger number of x and y.  
 */  
public int max(int x, int y){ ... }
```

### ► @param: Je Parameter, Beschreibung

```
@param name Beschreibung des Parameters.
```

### ► @return: Beschreibung des Rückgabewertes

```
@return The larger number of x and y.
```

### ► @throws: Geworfene Ausnahmen (später)

```
@throws ExceptionKlasse Beschreibung wann Ausnahme geworfen wird.
```

# JavaDoc-Tags

## Übergreifende JavaDoc-Tags

- ▶ **@deprecated**: Veraltete Elemente

```
@deprecated Has been superseded by ...
```

- ▶ **@since**: Version seitdem es das Element gibt

```
@since 1.0
```

- ▶ **@see**: Verweis auf anderes Element

```
@see de.hawlandshut.java1.oop.shapes.Point2D
```

Element kann sein:

- ▶ Klasse, **enum**, Interface
- ▶ Package

```
@see de.hawlandshut.java1.oop.shapes
```

- ▶ **Variable** mit # referenziert

```
@see java.lang.Math#PI  
@see #variableOfThisClass
```

- ▶ **@see**: Verweis auf anderes Element
  - ▶ **Konstruktor, Methode** mit **#** referenziert und Parametertypen

```
@see de.hawlandshut.java1.oop.shapes.Point2D #move(int, int)  
@see de.hawlandshut.java1.oop.shapes.Point2D #Constructor(int, int)  
@see #methodOfThisClass(double,double)
```

# Inline Javadoc-Tags

- ▶ Javadoc-Tags die innerhalb des Dokumentations-Textes verwendet werden
- ▶ Werden mit { } eingeschlossen
- ▶ Auswahl wichtiger Tags

Tag	Bedeutung
<code>{@inheritdoc}</code>	Erbe Beschreibung von Basisklasse/Interface
<code>{@link ref}</code>	Link auf anderes Element (s. @see)
<code>{@code c}</code>	Code-Schnipsel
<code>{@value s}</code>	Zeigt Wert eines statischen Felds

## ▶ Beispiele

```
Returns the maximum ({@see #min for the minimum})
```

```
@param obj Object to work with (must not be {@code null})
```

# Inhalt

## Dokumentation mit javadoc

### Erstellen der Dokumentation

## Beispiel

- ▶ Ausführliches Beispiel: `shapes/Rhombus.java`
- ▶ javadoc (im Verzeichnis examples ausgeführt)

```
javadoc  
-private \  
-version \  
-doctitle "Shapes Library" \  
-d doc \  
src/main/java/de/hawlandshut/java1/ oopbasics/shapes/*.java
```

- ▶ `-private`: bis zu **private**-Sichtbarkeit berücksichtigen (default: `-protected`)
- ▶ `-version`: `@version`-Tag berücksichtigen (default: aus)
- ▶ `-doctitle`: Titel der Dokumentation
- ▶ `-d`: Zielverzeichnis für HTML-Dateien
- ▶ Java-Dateien: Quelldateien
- ▶ Ergebnis: `index.html` (nicht vollständig dokumentiert)

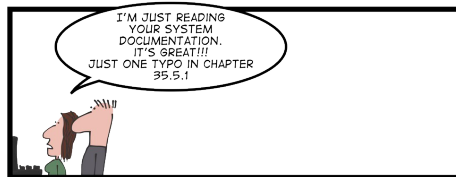
# Inhalt

## Dokumentation mit javadoc Ergänzungen



# Allgemeine: Hinweise zu Dokumentation

- ▶ Was sollte **dokumentiert** werden?
  - ▶ Alles was ein Entwickler wissen muss, wenn er die **Klassen verwenden** will!
- ▶ Was **muss nicht** dokumentiert werden?
  - ▶ Offensichtliche Methoden wie **Getter/Setter**, Kopier-Konstruktor; siehe (Negativ-)Beispiel `shapes/Rhombus.java`
  - ▶ **Überschriebene Methoden** (Dokumentation mit `{@inheritdoc}` erben)
- ▶ Trotz **JavaDoc**: Code kommentieren!



BE AWARE!!!



geek & poke



SOMEBODY MAY ACTUALLY READ IT!

# Doclets und Alternativen

- ▶ javadoc-Ausgabe wird über **Doclets** implementiert
  - ▶ **Standard Doclet**: Generiert HTML
  - ▶ **Doccheck**: Prüft Dokumentation (z.B. auf Vollständigkeit)
  - ▶ Mehr Doclets: <http://doclet.com/>
- ▶ Alternative zu javadoc: **doxygen**
  - ▶ <http://www.doxygen.nl/>
  - ▶ Funktioniert auch mit **anderen Sprachen**