

# Programmieren II: Java

Objektorientierte Programmierung in Java

Prof. Dr. Christopher Auer

Sommersemester 2024



Packages

Vererbung

Interfaces

Geschachtelte Typen

# Inhalt

## Packages

Idee hinter Packages

Packages deklarieren

Import von Paketen und Klassen

Statische Imports

Klassenpfad

jar -Dateien

# Inhalt

## Packages

Idee hinter Packages

## Idee hinter Packages

- ▶ Große Softwareprojekte brauchen eine **Organisationsstruktur**
  - ▶ Software-**Module** (z.B. nach Funktion, Abhängigkeiten)
  - ▶ **Namensräume** um Namenskonflikte zu vermeiden (vgl. Namespaces in C++/C#)
  - ▶ Abgrenzung von externen **Softwarepaketen** (z.B. Klassen des JDK, externe Bibliotheken)
  - ▶ Möglichkeit für den Compiler/Laufzeitumgebung **selektiv Teile der Software** zu laden
- ▶ Jede Programmiersprache hat eigene Ansätze

Sprache	Organisationseinheit	Definiert durch...
C++/C#	Namensräume	Code
Python	Module	Dateinamen/Verzeichnisstruktur
Java	Packages	Code <b>und</b> Verzeichnisstruktur

5

## Packages in Java

- ▶ Java-**Package**
  - ▶ **Verzeichnis** mit **Java-Klassen** (.class-Dateien)
  - ▶ **Namensraum**, d.h. Klassennamen innerhalb eines Packages müssen **eindeutig** sein
- ▶ **Name** eines Packages
  - ▶ **Kleinbuchstaben**, **Ziffern**, durch **Punkte** getrennt
  - ▶ **Beispiele**

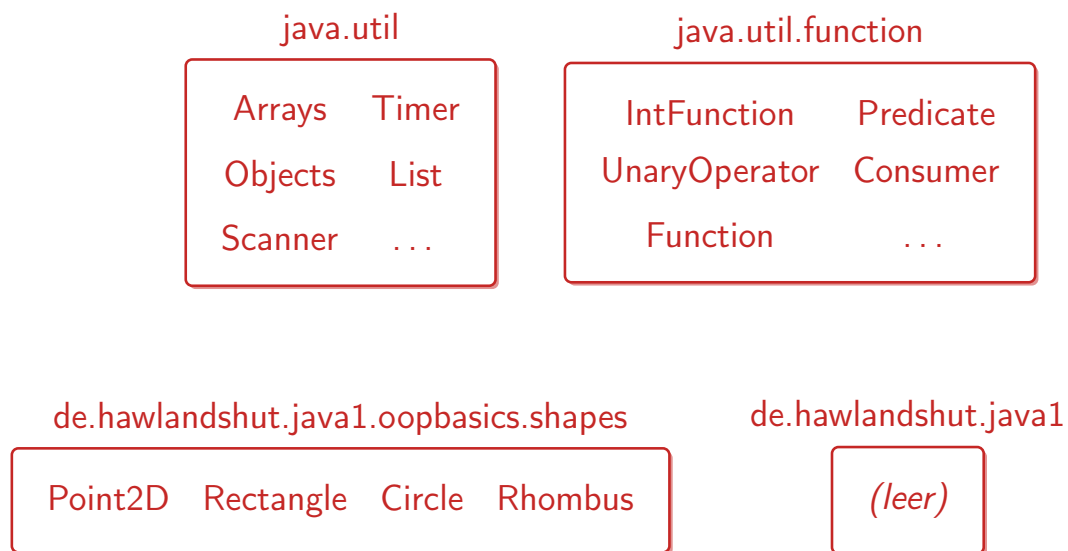
```
java
java.lang
java.util
javax.tools
de.hawlandshut.java1
de.hawlandshut.java1.oopbasics.shapes
```

- ▶ **Konvention**: wie umgekehrter DNS-Hostname

```
<top level domain>.<organization>
.<lib/programm name>.<package>.<sub-package>....
```

6

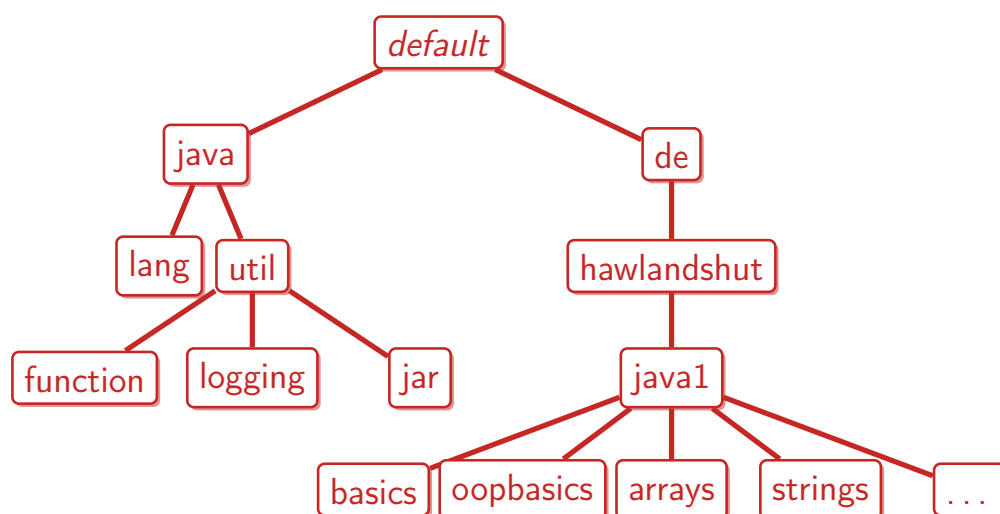
## Packages in Java: Beispiele



7

## Package-Hierarchie

- Packages bilden Hierarchie



- Hierarchie ist reine Organisationsstruktur
- Für Java liegen alle Pakete gleichberechtigt nebeneinander
- Packagepfad: Pfad von Wurzel aus, z.B. `java.util.jar`

8

## Packages und Verzeichnisstruktur

- ▶ JVM lädt Klassen beim **ersten Zugriff**
- ▶ **Schneller Start** der JVM
- ▶ Bei **erster Verwendung**: .class-Datei muss **schnell gefunden** werden
- ▶ Beispiel

```
de.hawlandshut.java1.oopbasics.shapes.Point2D
```

Hinweis: Hier **voll qualifizierter Klassenname**

- ▶ **ClassLoader** sucht und lädt .class-Datei

```
de/hawlandshut/java1/oopbasics/shapes/Point2D.class
```

- ▶ Pfad **relativ** zu **Classpath** (später)
- ▶ **Implikation**: Der **Packetpfad** muss dem **Verzeichnispfad** entsprechen
- ▶ Klasse Example im Paket p1.p2.p3...pn muss in **folgender Datei** zu finden sein

```
p1/p2/p3/.../pn/Example.class
```

## Inhalt

### Packages

Packages deklarieren

# Package-Deklaration

```
package de.hawlandshut.java1.oop;
```

- ▶ **Deklaration** über **package**
  - ▶ Klasse befindet sich im **Paket** de.hawlandshut.java1.oop
  - ▶ Liegt im **Verzeichnis** de/hawlandshut/java1/oop
- ▶ **package-Deklaration**
  - ▶ Am **Anfang** (vor **import**)
  - ▶ Nur **eine package**-Deklaration
  - ▶ Es darf nur **eine** Klasse mit dem gleichen **Bezeichner** im selben Paket geben

11

## Das Default-Package

```
public class Test { }
```

- ▶ Bei fehlender **package**-Deklaration liegt Klasse im **Default-Package**
- ▶ Default-Package ist **unbenannt**
- ▶ **Nicht-Default-Packages** sind **benannt** (z.B. java.util)
- ▶ **Hinweis:** Eine Klasse im Default-Package
  - ▶ kann **nur von Klassen im Default-Package** gesehen werden

```
public class TestUser{  
    private Test test;  
}
```

- ▶ kann **nicht von Klassen in benannten Packages** gesehen werden

```
package de.hawlandshut.java1.oop;  
public class TestUserInPackage{  
    private Test test; // FEHLER  
}
```

12

## Das Default-Package: Wann verwenden?

```
public class Test { }
```

- ▶ Default-Package: Wann verwenden?
  - ▶ Testprogramme („Ich probier mal schnell was aus“, d.h. nicht JUnit-Tests!)
  - ▶ Praktikum, Prüfung (außer wenn explizit verlangt)
- ▶ Default-Package nicht verwenden
  - ▶ im professionellen Betrieb
  - ▶ wenn Pakete erstellt und ausgeliefert werden sollen

13

## Inhalt

### Packages

Import von Paketen und Klassen

import

Wildcard-Import

14

## Motivation

- ▶ Per Default sieht zunächst eine Klasse nur die Klassen im **eigenen Paket**
- ▶ Grund: Compiler/JVM muss nicht **alle verfügbaren** Klassennamen **laden** und **halten**
- ▶ Zugriff auf Klasse in **anderem Paket**

```
var p =  
    new de.hawlandshut.java1.oopbasics.shapes.Point2D(1,2);
```

- ▶ **Voll qualifizierter Klassenname**: de.hawlandshut.java1.oopbasics.shapes.Point2D
- ▶ **Allgemein**: packagepfad.KlassenName
- ▶ Problem
  - ▶ unübersichtlich
  - ▶ viel **Schreibarbeit** (auch mit Autovervollständigung)
  - ▶ Ohne **var** wäre es **noch länger**
- ▶ „Lösung“: IDE sagen, sie soll **import** hinzufügen

15

## Inhalt

### Packages

Import von Paketen und Klassen

import

Wildcard-Import

16



## import

- Lösung: Klassenname **sichtbar machen** über **import**

```
import de.hawlandshut.java1.oopbasics.shapes.Point2D;
```

- Jetzt kürzer

```
var p = new Point2D(1,2);
```

- Ach, ja: Es gibt auch eine Klasse **java.awt.geom.Point2D**
- Was passiert wenn man **beide** Klassennamen importiert?

```
import de.hawlandshut.java1.oopbasics.shapes.Point2D;  
import java.awt.geom.Point2D; FEHLER
```

- „A type with the same simple-name is already defined“
- Erkenntnisse
  1. Geht nicht (**Namenskonflikt** )
  2. Point2D nennt man **simple-name** (im Vergleich zum **qualifizierten voll Namen** )

17

## Namenskonflikte

```
import de.hawlandshut.java1.oopbasics.shapes.Point2D;  
import java.awt.geom.Point2D; FEHLER
```

- Was tun in diesem Fall?
- Voll **qualifizierten Namen** verwenden
- Zumindest für **eine Klasse**

```
import de.hawlandshut.java1.oopbasics.shapes.Point2D;  
  
public class Test{  
    private Point2D hawPoint;  
    private java.awt.geom.Point2D awtPoint;  
}
```

- Aber: Situation kommt **sehr selten** vor

18

# Inhalt

## Packages

### Import von Paketen und Klassen

`import`

Wildcard-Import

19

## Wildcard-Import

- ▶ Manchmal verwendet man **viele** Klassen eines Pakets

```
import de.hawlandshut.java1.oopbasics.shapes.Point2D;  
import de.hawlandshut.java1.oopbasics.shapes.Rectangle;  
import de.hawlandshut.java1.oopbasics.shapes.Circle;  
import de.hawlandshut.java1.oopbasics.shapes.Rhombus;  
...
```

- ▶ **Viele** `import`-Anweisungen
- ▶ **Alternative:** Wildcard-Import

```
import de.hawlandshut.java1.oopbasis.shapes.*;
```

- ▶ Macht **alle Klassen** in diesem Paket sichtbar
- ▶ Aber **nicht** die von **Unterpaketen**!

20

## Verwendung von Wildcard-Imports

- ▶ Gültig
  - ▶ ein Wildcard
  - ▶ am Ende des Paketpfads

- ▶ Ungültige Beispiele

```
import de.hawlandshut.java1.*.*;  
import *.util;
```

- ▶ Hinweise zur Verwendung
  - ▶ Besser Imports einzelner Klassen
  - ▶ Durch IDE verwalten lassen
  - ▶ `import package.*`-Deklaration verweisen oft
- ▶ Noch ein Grund für Einzel-Imports:
  - ▶ `import`-Anweisungen geben Abhängigkeiten an
  - ▶ Viele `import`-Anweisungen, viele Abhängigkeiten
  - ▶ Indiz für zu komplexe Klasse

21

## Wildcard-Imports und Namenskonflikte

- ▶ Beispiel: Doppelter Import von Point2D

```
import java.awt.geom.*;  
import de.hawlandshut.java1.oopbasics.shapes.*;
```

- ▶ Bisher kein Fehler
- ▶ Aber erste Verwendung führt zu Fehler

```
Point2D p = new Point2D(); FEHLER
```

- ▶ Lösung: Voll qualifizierter Namen verwenden
- ▶ Oder: zu verwendende Klasse einzeln importieren

```
import java.awt.geom.*;  
import de.hawlandshut.java1.oopbasics.shapes.Point2D;
```

```
Point2D p = new Point2D(); // de.hawlandshut...
```

22

# Wildcard-Imports und Namenskonflikte

- **Hinweis:** Klassen im **eigenen Paket** haben Vorrang

```
package de.hawlandshut.java1.oopbasics.shapes;  
import java.awt.geom.*;  
  
public class Test{  
    // de.hawlandshut.java1.oopbasics.shapes.Point2D  
    private Point2D point = new Point2D();  
}
```

23

## Inhalt

### Packages

#### Statische Imports

Statischer Import von enum-Werten

Statische Imports: Hinweise

24

## Statische Imports

- **Zur Erinnerung:** Verwendung von statischen Attribute/Methoden

```
double r = Math.sin(Math.PI/4);  
System.out.println(r);
```

### Statische Elemente

- [Math.sin](#), [Math.PI](#)
- [System.out](#)
- Keine **Instanz** nötig, dafür immer der **Klassenname**
- Oft viel **Schreibarbeit**

```
Math.pow(Math.sin(Math.PI/2),2)  
+ Math.pow(Math.sin(Math.PI/4),2)
```

Sechsmal [Math](#)!

25

## Statische Imports

```
Math.pow(Math.sin(Math.PI/2),2)  
+ Math.pow(Math.sin(Math.PI/4),2)
```

- **Abhilfe:** Statische Imports

```
import static java.lang.Math.sin;  
import static java.lang.Math.pow;  
import static java.lang.Math.PI;
```

```
pow(sin(PI/2),2) + pow(sin(PI/4),2)
```

- **Allgemeine Syntax:**

```
import static packagename.Klasse.statischesElement;  
macht „statischesElement“ als simple-name sichtbar
```

- Sogar **Wildcard** möglich

```
import java.lang.Math.*;
```

26

# Inhalt

## Packages

### Statische Imports

Statischer Import von enum-Werten

Statische Imports: Hinweise

27

## Statischer Imports von enum-Werten

### ► Zur Erinnerung: **enums**

```
package de.hawlandshut.java1.oopbasics;  
public enum Weekday {  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY,  
    FRIDAY, SATURDAY, SUNDAY }  
}
```

### ► **enum**-Werte werden **transformiert** in

```
public class Weekday {  
    public static final Weekday MONDAY = new Weekday();  
    /* ... */  
}
```

### ► Damit können die **enum-Werte** statisch importiert werden

```
import static de.hawlandshut.java1.oopbasics.Weekday.*;
```

```
if (day == MONDAY){ /* ... */ }
```

28

## Statischer Imports von enum-Werten

```
import static de.hawlandshut.java1.oopbasics.Weekday.*;
```

- ▶ Aber: Folgender Code führt zu Fehler

```
Weekday day = MONDAY; // FEHLER
```

- ▶ „Unknown symbol Weekday“
- ▶ Grund
  - ▶ Nur die Werte von Weekday wurden importiert
  - ▶ Nicht die Klasse Weekday selbst
- ▶ Abhilfe

```
import de.hawlandshut.java1.oopbasics.Weekday;
```

## Inhalt

### Packages

#### Statische Imports

Statischer Import von enum-Werten

Statische Imports: Hinweise

## Statische Imports: Hinweise


- Lokale **statische Methoden/Attribute** können statische Imports **verschatten**

```
4 import static java.lang.Math.sin;
5 import static java.lang.Math.PI;
```

StaticImportExamples.java

```
12 private static double sin(double x){
13     // a good approximation (for small angles...)
14     return x;
15 }
```

StaticImportExamples.java

```
21  runStaticImportConflictExample
22 System.out.printf("sin(PI/2)=%f%n", sin(PI/2));
```

StaticImportExamples.java

```
sin(PI/2)=1,570796
```

31

## Statische Imports: Hinweise

- Immerhin: IDE **warnt**, dass statischer Import **nicht verwendet** wird

```
import static java.lang.Math.sin;
```

- Aber bei

```
import static java.lang.Math.*;
```

warnt sie **nicht mehr!**

- Erkenntnisse
  - Statische Einzel-**imports** mit **Vorsicht** verwenden
  - Statische Wildcard-**imports** **nie** verwenden

32



## Klassenpfad: Definition

- ▶ Klassenpfad („classpath“)
  - ▶ Liste von **Verzeichnispfaden** in denen der Classloader nach Klassen sucht
  - ▶ Über **Kommandozeile**

```
java/c -cp pfad1:pfad2:...:pfadN
```

- ▶ Über **Umgebungsvariable**

```
CLASSPATH="pfad1:pfad2:...:pfadN"
```

- ▶ **Format**

- ▶ Pfad kann **relativ** oder **absolut** sein
- ▶ **Linux/macOS/Unix**: Trennzeichen :

```
pfad1:pfad2:...:pfadN
```

- ▶ **Windows**: Trennzeichen ;

```
pfad1;pfad2;...;pfadN
```

## Klassenpfad: Beispiel

- Java-Programm: Erstellt `p=Point2D`, verschiebt `p` und gibt `p` aus

```
2 package de.hawlandshut.java1.oop;  
4 import de.hawlandshut.java1.oopbasics.shapes.Point2D;  
6 public class Point2DExample {  
8     public static void main(String args[]){  
9         Point2D p = new Point2D(1,2);  
10        p.move(3,2);  
11        System.out.println(p);  
12    }  
14 }
```

📄 Point2DExample.java

35

## Klassenpfad: Beispiel

- Übersetzen (1. Versuch)

```
% javac src/main/java/de/hawlandshut/java1/oop/ Point2DExample.java  
error: cannot find symbol Point2D
```

**Problem:** javac kann `Point2D.class` nicht finden

- Point2D übersetzen

```
% javac src/main/java/de/hawlandshut/java1/oopbasics/shapes/ Point2D.java
```

Erzeugt `src/main/.../shapes/Point2D.class`

- Übersetzen (2. Versuch)

```
% javac src/main/java/de/hawlandshut/java1/oop/ Point2DExample.java  
error: cannot find symbol Point2D
```

**Problem:** javac weiß nicht wo er nach `Point2D` suchen soll

36

## Klassenpfad: Beispiel

### ► Übersetzen: (3. Versuch)

```
% javac -cp src/main/java src/main/java/de/hawlandshut/java1/oop/ ↵  
Point2DExample.java
```

Erzeugt: Point2DExample.class

Klassenpfad:

- src/main/java
- **Compiler** sucht nach Point2D in

src/main/java/ de/hawlandshut/java1/oopbasics/shapes/...  
Klassenpfad                      Paketpfad  
Point2D.class  
Klassenname

### ► javac sucht in **allen** Klassenpfad

37

## Klassenpfad: Beispiel

### ► Ausführen: (1. Versuch)

```
% java de.hawlandshut.java1.oop.Point2DExample  
Error: Could not find or load main class de.hawlandshut.java1.oop.Point2DExample
```

**Problem:** java weiß nicht wo Point2DExample zu suchen ist

### ► Ausführen: (2. Versuch)

```
% java -cp src/main/java de.hawlandshut.java1.oop.Point2DExample  
Point2D: { x = 4, y = 4 }
```

Klassenpfad src/main/java

- Point2DExample kann gefunden werden
- **Und:** Point2D kann gefunden werden

### ► Äquivalent mit CLASSPATH (hier Unix)

```
% export CLASSPATH="src/main/java"  
% java de.hawlandshut.java1.oop.Point2DExample  
Point2D: { x = 4, y = 4 }
```

38

# Inhalt

## Packages

jar -Dateien

39

## jar -Dateien

- ▶ **jar**: „Java Archive“
  - ▶ **zip**-Datei mit zusätzlichen Informationen
  - ▶ Enthält „Java-Ressourcen“, vor allem **.class-Dateien**
  - ▶ Können wie **Klassenpfad** verwendet werden
  - ▶ Oft für **externe APIs/Libraries**/etc.
- ▶ **Vorteile**
  - ▶ **Kompakt**: eine komprimierte Datei statt vieler
  - ▶ Meta-Informationen
- ▶ **Nachteile**
  - ▶ Muss beim Laden **dekomprimiert** werden
  - ▶ Kein **direkter Zugriff** sondern über [↗](#) **ClassLoader**

40

# jar-Tool

- ▶ jar-Tool zum Erstellen/Anzeigen/Extrahieren
  - ▶ Aufrufparameter ähnlich wie tar (Unix/Linux)
  - ▶ Erstellen

```
jar -cf <jar-Datei> datei1 datei2 ...
```

- ▶ Inhalt anzeigen

```
jar -tf <jar-Datei>
```

- ▶ Extrahieren

```
jar -xf <jar-Datei> [datei1 datei2 ...]
```

- ▶ -v: ausführliche Ausgabe
- ▶ -e: „entry-class“ angeben (main-Methode)
- ▶ -help: Hilfe anzeigen

41

## jar-Datei: Beispiel

- ▶ Erstellen einer jar-Datei

```
% jar -cf shapes.jar de/hawlandshut/java1/oopbasics/shapes/*.class
```

Hinweis: Relativer Pfad muss mit **Klassenpfad** übereinstimmen

- ▶ Inhalt anzeigen

```
% jar -tf shapes.jar
de/hawlandshut/java1/oopbasics/shapes/Point2D.class
de/hawlandshut/java1/oopbasics/shapes/Circle.class
...
```

42

## jar-Datei: Beispiel

### ► jar-Datei beim **Kompilieren**

```
% javac -cp shapes.jar src/main/java/de/hawlandshut/java1/oop/ ↵  
    Point2DExample.java
```

**Hinweis:** jar-Datei im **Klassenpfad**

### ► jar-Datei beim **Ausführen**

```
% java -cp shapes.jar:src/main/java de.hawlandshut.java1.oop.Point2DExample  
Point2D: { x = 4, y = 4 }
```

### **Klassenpfad**

- shapes.jar für Point2D
- src/main/java für Point2DExample

43

## Inhalt

### **Vererbung**

- Motivation
- Begleitendes Beispiel
- Vererbung mit extends
- Die Mutter aller Klassen: Object
- Konstruktoren in Hierarchien
- Typen in der Hierarchie
- Überschreiben von Methoden
- Überschreiben der Methoden von Object
- Finale Methoden und Klassen
- Dynamische und statische Bindung
- Abstrakte Klassen und Methoden
- Zusammenfassung

44

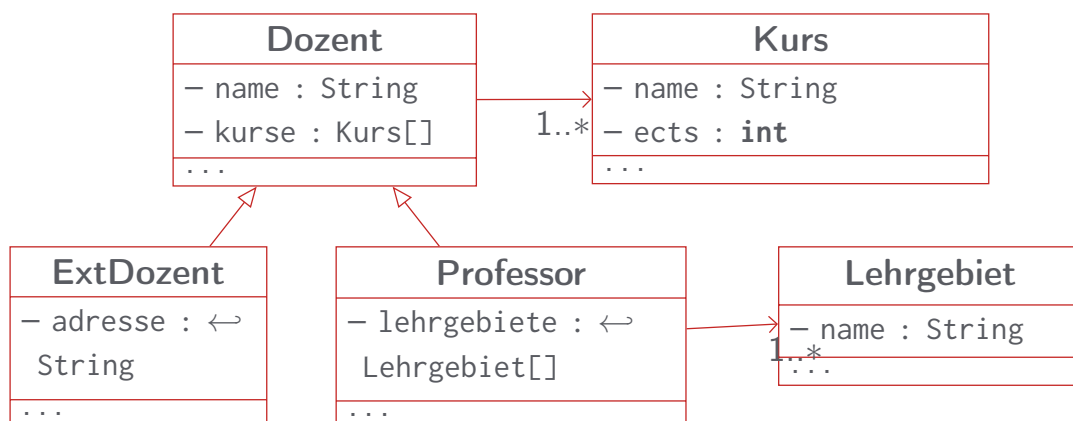
# Inhalt

## Vererbung Motivation

45

## Motivation

- ▶ Klassen und Objekte modellieren **Beziehungen** zwischen Dingen
- ▶ Arten von **Beziehungen**
  - ▶ „**hat**“-Beziehung: Dozent **hat** Kurse
  - ▶ „**ist ein**“-Beziehung:
    - ▶ Externer Dozent **ist ein** Dozent
    - ▶ Professor **ist ein** Dozent
  - ▶ **Kombination**: Professor **ist ein** Dozent und **hat** Lehrgebiete

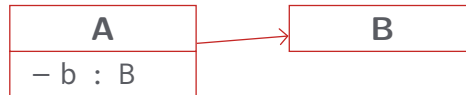


46

## Motivation

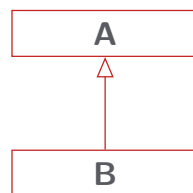
### ► „hat“-Beziehung:

- Assoziation
- Java: Klasse **hat** Referenz auf anderes Objekt
- UML: A **hat ein** B



### ► „ist ein“-Beziehung:

- Vererbung (dieses Kapitel)
- Java: Klasse **leitet** von anderer Klasse ab
- (Später: **interfaces**)
- UML: B **ist ein** A



47

## Motivation

### ► Man sagt

- B **erbt** von A/B **leitet** von A **ab**/B **erweitert** A
- A ist die **Basis-/Ober-/Eltern-/Superklasse** von B
- B ist eine **Sub-/Unter-/Kind-/Erweiterungsklasse** von A

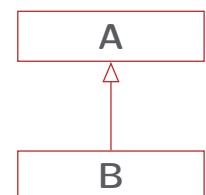
### ► A vererbt seine **Eigenschaften** an B

### ► Sichtbare Eigenschaften von A hat somit auch B

- Dozent hat **Kurse**
- Somit: **externe Dozenten** und **Professoren** haben auch Kurse

### ► B kann A um **Eigenschaften** erweitern

- Professor hat **Lehrgebiete**
- Externer Dozent hat **Adresse**

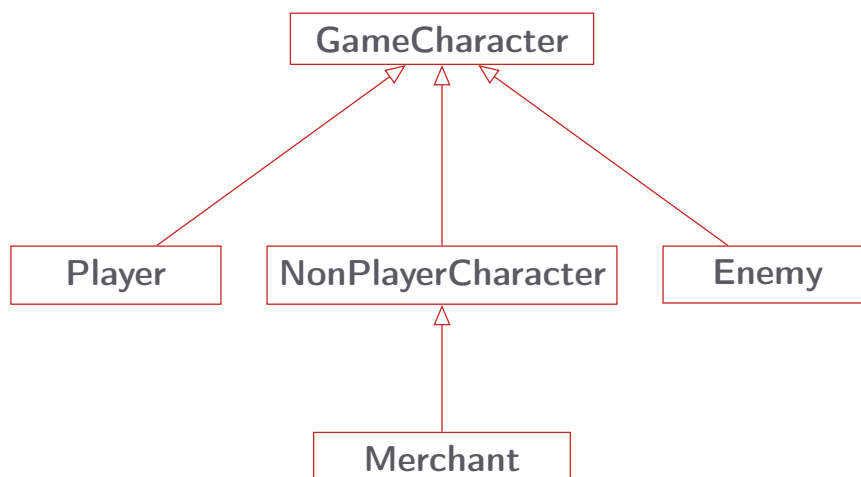


48



## Computer-Rollenspiel

- ▶ Beispiel für die nächsten Kapitel
- ▶ Character eines (**sehr vereinfachten**) Computer-Rollenspiels
- ▶ Wird Schritt für Schritt erweitert



# Inhalt

## Vererbung

Vererbung mit extends

51

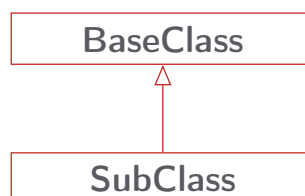
## Vererbung in Java

- ▶ Schlüsselwort **extends**
- ▶ Einfachvererbung, **keine** Mehrfachvererbung
- ▶ Allgemeine Syntax

```
class SubClass extends BaseClass
```

SubClass **erweitert** BaseClass

- ▶ UML



52

## Beispiel: GameCharacter

- ▶ **GameCharacter**: Basisklasse aller Character in unserem Spiel

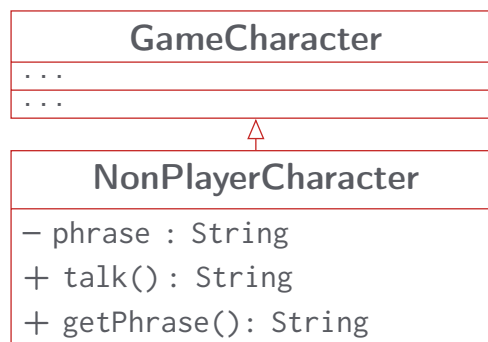
GameCharacter
- name : String - health : <b>int</b> - x : <b>int</b> - y : <b>int</b>
+ getName(): String + getHealth(): <b>int</b> + isAlive(): <b>boolean</b> + getX(): <b>int</b> + getY(): <b>int</b> + changeHealth(amount : <b>int</b> ) # move(dx : <b>int</b> , dy : <b>int</b> )

- ▶ Implementierung: `game/GameCharacter.java`
- ▶ Hinweis: Manche Elemente der Implementierung werden erst **später** eingeführt

53

## Beispiel: NonPlayerCharacter

- ▶ NonPlayerCharacter soll GameCharacter **erweitern**



54

## Beispiel: NonPlayerCharacter

### ► Deklaration

```
9 public class NonPlayerCharacter extends GameCharacter
```

game/NonPlayerCharacter.java

### ► Attribut phrase: Was der Character zu **sagen** hat (**final**)

### ► Methode talk(): Lässt den Character **sprechen**

```
33 public String talk() {  
34     return String.format("Hello, my name is %s! %s",  
35         getName(), phrase);  
36 }
```

game/NonPlayerCharacter.java

55

## Beispiel: NonPlayerCharacter

### ► Was **sieht** NonPlayerCharacter von GameCharacter?

### ► Sichtbar

- **public**: getName(), getHealth(), isAlive(), getX/Y()
- **protected**: move()
- **Paket-sichtbar**: hier nichts

### ► Nicht sichtbar

- **private**: name, health, x, y

### ► Zugriff auf **private**-Attribute/Methoden würde zu Fehler führen

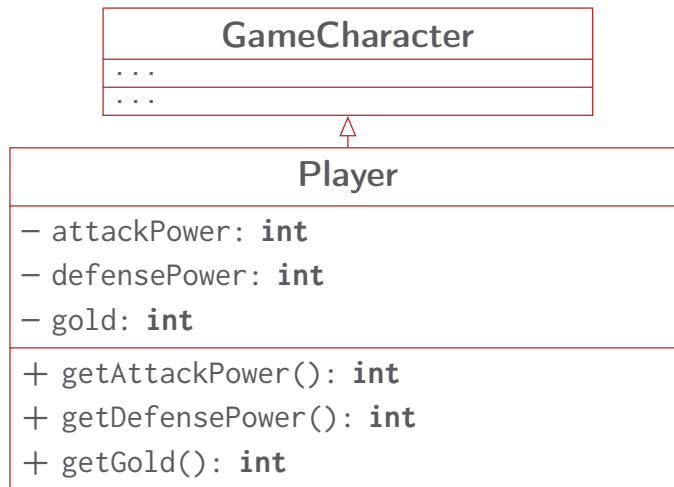
```
public String talk() {  
    return String.format("Hello, my name is %s! %s",  
        name, phrase); // FEHLER  
}
```

### ► „The field name is not visible.“

56

## Beispiel: Player

- ▶ Player modelliert den Spieler unseres Rollenspiels
- ▶ Player erweitert **ebenfalls** GameCharacter



- ▶ Code

11 **public class** Player **extends** GameCharacter

game/Player.java

57

## Beispiel: Player

- ▶ Was **erbt** Player von GameCharacter?
  - ▶ Player hat einen **Namen**
  - ▶ Player hat eine **Position**
  - ▶ Player hat einen **Gesundsheitswert**
- ▶ Zusätzlich **erweitert** Player GameCharacter um
  - ▶ Offensiv-/Defensivwerte
  - ▶ Goldmünzen

player.getName();

player.getX(); player.getY();

player.getHealth();

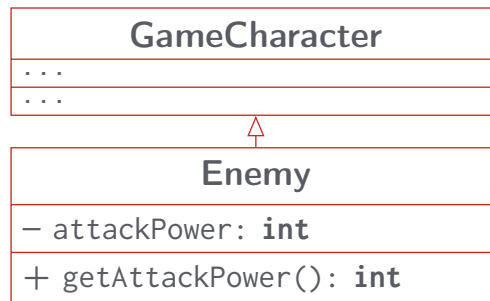
player.getAttackPower();  
player.getDefensePower();

player.getGold();

58

## Beispiel: Enemy

- ▶ Enemy modelliert für Gegner des Spielers, z.B., ein Drache
- ▶ Enemy erweitert **ebenfalls** GameCharacter

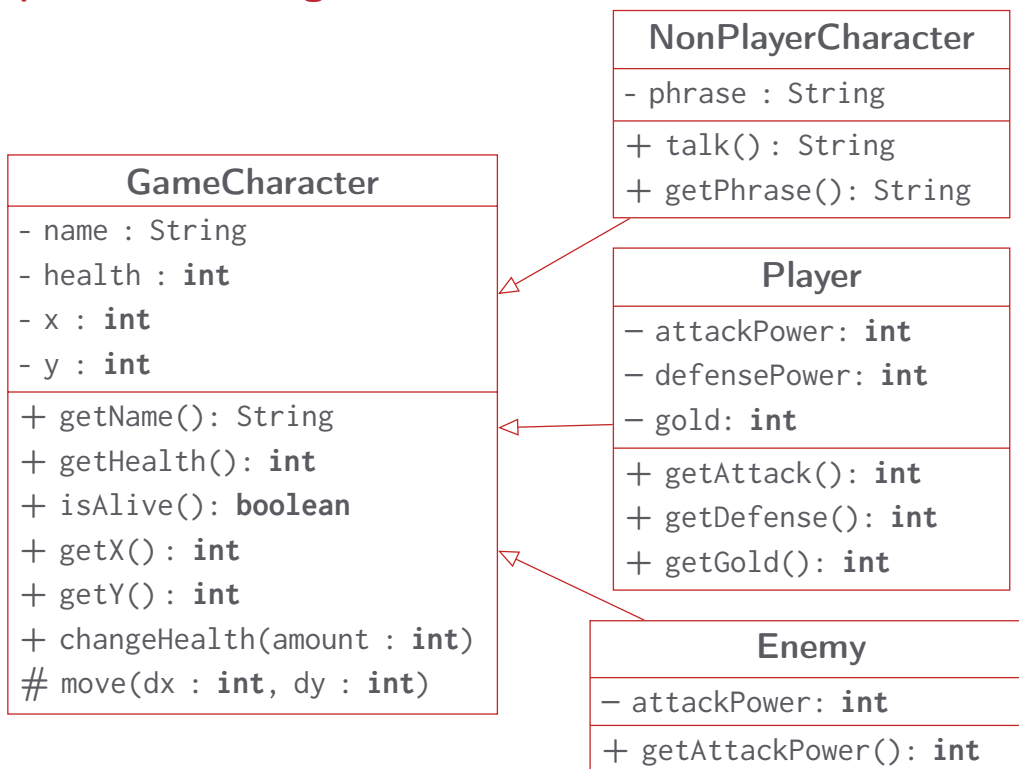


- ▶ Code

```
public class Enemy extends GameCharacter
```

59

## Beispiel: Klassendiagramm bisher



60

# Inhalt

## Vererbung

Die Mutter aller Klassen: Object

61

## Die Mutter aller Klassen: Object

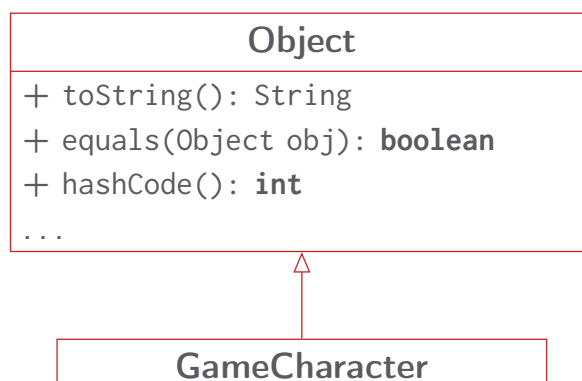
- ▶ Was passiert wenn man **extends** weglässt?

```
public class GameCharacter
```

- ▶ Dann leitet GameCharacter **implizit** von java.lang.Object ab


```
public class GameCharacter extends Object
```

- ▶ [↗](#) Object vererbt grundlegende Methoden an **alle Klassen**



62

## Methoden der Klasse Object

- ▶ `toString()` — liefert  `String`-Repräsentation (sollte überschrieben werden)
- ▶ `equals(Object obj)` — prüft auf Identität (Gleichheit nur bei Überschreibung; später)
- ▶ `hashCode()` — liefert Hashwert (sollte überschrieben werden)
- ▶ `getClass()` — gibt den Objekttyp zurück
- ▶ `clone` — kopiert das Objekt (muss von Subklassen überschrieben werden; später)
- ▶ `notify`, `notifyAll`, `wait` — für Multi-Thread-Programmierung (Programmieren III)

63

## Inhalt

### Vererbung

Konstruktoren in Hierarchien

64



## Konstruktoren von GameCharacter

- ▶ GameCharacter hat einen **Initialisierungs-Konstruktor**

```
25 public GameCharacter(String name, int health, int x, int y) {  
26     this.name = name;  
27     this.health = health;  
28     this.x = x;  
29     this.y = y;  
30 }
```

game/GameCharacter.java

- ▶ GameCharacter hat einen **Kopier-Konstruktor**

```
37 public GameCharacter(GameCharacter gameCharacter){  
38     this.name = gameCharacter.getName();  
39     this.health = gameCharacter.getHealth();  
40     this.x = gameCharacter.getX();  
41     this.y = gameCharacter.getY();  
42 }
```

game/GameCharacter.java

65

## Konstruktor von NonPlayerCharacter

- ▶ NonPlayerCharacter soll folgenden Konstruktor besitzen

```
public NonPlayerCharacter(String name, String phrase, int x, int y)
```

- ▶ health-Wert soll auf 1 initialisiert werde
- ▶ **name**, **phrase** und **x/y** werden über Parameter initialisiert
- ▶ **Problem**: NonPlayerCharacter kann name und Position nicht **setzen**, da **private** in GameCharacter

```
public NonPlayerCharacter(String name, String phrase, int x, int y){  
    this.name = name; // FEHLER  
    this.x = x; // FEHLER  
    this.y = y; // FEHLER  
    this.phrase = phrase;  
}
```

- ▶ Nur über **Konstruktor** von **GameCharacter** möglich
- ▶ Wie können wir diesen Konstruktor **aufrufen**?

66

## Konstruktor von NonPlayerCharacter

- Zur Erinnerung: Konstruktoren der **eigenen** Klasse kann man mit **this** aufrufen

```
public Point2D(int x, int y){  
    this.x = x;  
    this.y = y;  
}
```

```
public Point2D(){  
    this(0,0);  
}
```

- Gleiche Idee: Konstruktor der **Basisklasse** mit **super** aufrufen

```
15 public NonPlayerCharacter(String name, String phrase, int x, int y) {  
16     super(name, 1, x, y); // health is 1  
17     this.phrase = phrase;  
18 }
```

game/NonPlayerCharacter.java

67

## Kopier-Konstruktor von NonPlayerCharacter

- NonPlayerCharacter soll **Kopier-Konstruktor** haben
- GameCharacter definiert bereits Kopier-Konstruktor
- Frage: Erbt NonPlayerCharacter den Kopier-Konstruktor?
- (Leider) nein

```
var yennefer = new NonPlayerCharacter("Yennefer", "*sigh*", 0, 0);  
var yenneferClone = new NonPlayerCharacter(yennefer); // FEHLER
```

„The constructor is undefined“

- Erkenntnis: Konstruktoren werden **nicht** vererbt
- Implementierung

```
22 public NonPlayerCharacter(NonPlayerCharacter other){  
23     super(other);  
24     this.phrase = other.getPhrase();  
25 }
```

game/NonPlayerCharacter.java

68

## Hinweise zu **super**

- ▶ **Hinweis:** **super**-Konstruktoraufruf muss **erste Anweisung** sein

```
public NonPlayerCharacter(NonPlayerCharacter other){  
    this.phrase = other.getPhrase();  
    super(other); // FEHLER  
}
```

69

## Konstruktoren von **Player**

- ▶ Player soll folgenden Konstruktor haben
  - ▶ Player(**int** gold, **int** attackPower, **int** defensePower)
  - ▶ Werte von gold, attackPower, defensePower werden übernommen
  - ▶ **Name** ist „Geralt von Riva“
  - ▶ **Health** soll 100 sein
  - ▶ **Position** soll x=0, y=0 sein
- ▶ Implementierung

```
45 public Player(int gold, int attackPower, int defensePower) {  
46     super("Geralt von Riva", 100, 0, 0);  
47     this.gold = gold;  
48     this.attackPower = attackPower;  
49     this.defensePower = defensePower;  
50 }
```

game/Player.java

70

## Konstruktoren von Player

► Player braucht auch einen **Default-Konstruktor**

- Attribute von GameCharacter wie oben
- Gold: 0
- attack/defensePower: 1/1

► Implementierung:

```
55 public Player(){
56     this(0,1,1);
57 }
```

game/Player.java

► **Konstruktor-Aufrufe**

1. Player() → **this**(0,1,1)
2. Player(0,1,1) → **super**("Gerald von Riva", 100, 0, 0)
3. GameCharacter("Gerald von Riva", 100, 0, 0)

71

## Konstruktoren von Player

► Und noch ein **Kopier-Konstruktor** für Player

```
public Player(Player other){
    this.gold = other.getGold();
    this.attackPower = other.getAttackPower();
    this.defensePower = other.getDefensePower();
}
```

► **Fehler:** „No suitable constructor found for GameCharacter(no argument)“ ?

► **Beobachtung:** Es **fehlt super**

► **Impliziter** Aufruf von Default-Konstruktor der Basisklasse

```
public Player(Player other){
    super(); // FEHLER
    this.gold = other.getGold();
    this.attackPower = other.getAttackPower();
    this.defensePower = other.getDefensePower();
}
```

► GameCharacter hat aber **keinen** Default-Konstruktor

72

## Konstruktoren von Player

- **Besser:** Kopier-Konstruktor von GameCharacter aufrufen

```
61 public Player(Player other){
62     super(other);
63     this.gold = other.getGold();
64     this.attackPower = other.getAttackPower();
65     this.defensePower = other.getDefensePower();
66 }
```

game/Player.java

- **Erkenntnisse**

- Fehlt **super**-Aufruf wird **Default-Konstruktor** aufgerufen
- Hat Basisklasse **keinen Default-Konstruktor: FEHLER**
- Es **muss immer** ein Konstruktor der Basisklasse **explizit** oder **implizit** aufgerufen werden

73

## super-this-Konflikt

```
public Player(){
    super("Geralt von Riva", 100, 0, 0);
    this(0, 1, 1); // FEHLER
}
```

- **this** und **super** müssen **erste Anweisungen** sein
- **Alternativen:**
  - Aufruf von **this** mit impliziten Aufruf von **super** (s. oben)
  - Initialisierung-Methoden statt **this**

```
private initialize(int gold, int attackPower,
    int defensePower){
    this.gold = gold;
    this.attackPower = attackPower;
    this.defensePower = defensePower;
}
```

```
public Player(){
    super("Geralt von Riva", 100, 0, 0);
    initialize(0, 1, 1);
}
```

74

# Inhalt

## Vererbung

Typen in der Hierarchie

„ist-ein“-Beziehung

Widening Cast

Narrowing Cast

ClassCastException

instanceof-Operator

# Inhalt

## Vererbung

Typen in der Hierarchie

„ist-ein“-Beziehung

Widening Cast

Narrowing Cast

ClassCastException

instanceof-Operator

## „ist-ein“-Beziehung

### ► Ableitungshierarchie definiert „ist-ein“-Beziehung

- GameCharacter ist ein `Object`
- Player ist ein GameCharacter
- NonPlayerCharacter ist ein GameCharacter

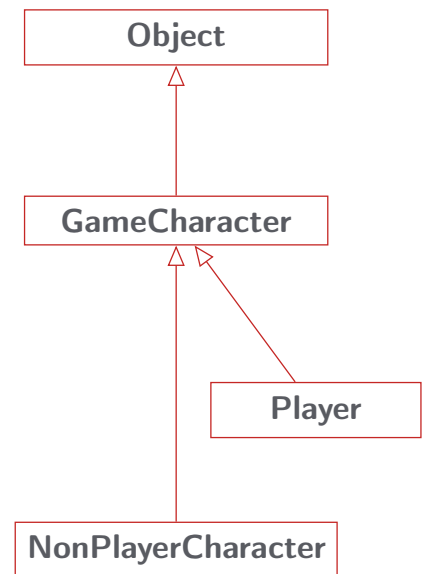
### ► Eigenschaften

- **Transitiv**: Player ist ein `Object`
- **Reflexiv**: Player ist ein Player
- **Antisymmetrisch**: „es gibt keine Zyklen“, wie

A **extends** B **extends** A

### ► Wir sagen


- Player ist **spezieller** als `Object`
- `Object` ist **allgemeiner** als Player
- Player und NonPlayerCharacter sind **inkompatibel**



77

## „ist-ein“-Beziehung in Java: Objekt- und Referenztyp

### ► Beispiel

```
12  runObjectTypeRefExample
13 GameCharacter character = new Player();
14 out.printf("character.getName(): %s\n",
15     character.getName());
```

 CastExamples.java

character.getName(): Gerecht von Riva

78

# Objekt- und Referenztyp

## ► Situation im Beispiel

`character : GameCharacter` → `Player : name=...`

## ► Referenztyp vs. Objekttyp

	Typ von	Beispiel
<b>Referenztyp</b>	Referenzvariable	GameCharacter
<b>Objekttyp</b>	Objekt	Player

79

## Inhalt

### Vererbung

Typen in der Hierarchie

„ist-ein“-Beziehung

Widening Cast

Narrowing Cast

ClassCastException

instanceof-Operator

80



## Impliziter Widening Cast bei Zuweisung


- ▶ Seien x und y Referenzvariablen vom Typ X und Y
- ▶ Dann kann x y **direkt zugewiesen** werden, d.h.

```
x = y;
```

- ▶ wenn X **allgemeiner** als Y ist (X=Y möglich)
- ▶ Beispiel

```
21 Player player = new Player();  
22 GameCharacter character = player;  
23 Object object = character;
```

CastExamples.java

- ▶ GameCharacter ist allgemeiner als Player
- ▶  Object ist allgemeiner als GameCharacter
- ▶ Java führt einen **impliziten widening Cast** durch

```
GameCharacter character = (GameCharacter) player;
```


81

## Impliziter Cast allgemein

- ▶ Substitutionsprinzip
  - ▶ Ist X **allgemeiner** als Y so...
  - ▶ kann eine Referenz vom Typ Y **überall** verwendet werden,...
  - ▶ wo eine Referenz vom Typ X **verwendet** wird.
- ▶ Beispiel

```
28 public static void printName(GameCharacter character) {  
29     out.printf("Name: %s\n", character.getName());  
30 }
```

CastExamples.java

```
35  runPrintNameExample  
36 Player player = new Player();  
37 printName(player);
```

CastExamples.java

- ▶ GameCharacter ist **allgemeiner** als Player
- ▶ player kann im Aufruf von printName verwendet werden
- ▶ Widening Cast erfolgt **implizit**

82

# Inhalt

## Vererbung

### Typen in der Hierarchie

„ist-ein“-Beziehung

Widening Cast

Narrowing Cast

ClassCastException

instanceof-Operator

83

## Narrowing Cast

- ▶ Umkehrung vom widening Cast: narrowing Cast

- ▶ Beispiel

```
Object object = new Player(); // widening  
printName(object); // FEHLER
```

„Incompatible types: Object cannot be converted to GameCharacter“

- ▶ Problem:
  - ▶ printName erwartet GameCharacter
  - ▶ GameCharacter ist **spezieller** als **Object**
  - ▶ **Narrowing Cast** nötig

84

## Cast-Operator für Referenztypen

### ► Cast-Operator (siehe Kapitel zu Operatoren)

#### ► Bisher für **primitive Type**

```
int i = 42;  
byte b = (int) i;
```


Narrowing Cast from „größeren zum kleineren Typen“

#### ► Jetzt Cast-Operator auf **Referenztypen**

```
X x = (X) y;
```

Narrowing Cast von Referenztyp von y auf Typ X


### ► Beispiel

```
46  runNarrowingCastExample  
47 Object object = new Player(); // widening  
48 printName((GameCharacter) object); // narrowing
```

 CastExamples.java

85

## Narrowing Cast: Beispiel

```
54  runNarrowingCastExample2  
55 Object object = new NonPlayerCharacter(  
56     "Yennefer", "...", 0, 0);  
58 // narrowing casts  
59 GameCharacter character = (GameCharacter) object;  
60 NonPlayerCharacter npc = (NonPlayerCharacter) object;  
62 out.printf("object.hashCode(): %d\n", object.hashCode());  
63 out.printf("character.getName(): %s\n", character.getName());  
64 out.printf("npc.talk(): %s\n", npc.talk());
```

 CastExamples.java

```
object.hashCode(): 1183374741  
character.getName(): Yennefer  
npc.talk(): Hello, my name is Yennefer! ...
```

86

## Ungültiger Narrowing Cast

- ▶ Warum darf ein widening Cast **implizit** sein?
- ▶ Warum muss der narrowing Cast **explizit** sein?
- ▶ Grund
  - ▶ Widening Cast **funktioniert immer**
  - ▶ Narrowing Cast **kann schiefgehen**

```
String s = "I'm a GameCharacter, I swear!";  
printName((GameCharacter) s); // FEHLER
```

„Incompatible types: String cannot be converted to GameCharacter“

- ▶ **String** und GameCharacter sind **inkompatibel**
- ▶ **Compiler** kann hier auf Inkompatibilität prüfen
- ▶ Aber das ist **nicht immer** möglich

87

## Narrowing Cast

- ▶ Beispiel

```
69 public static void letNPCTalk(  
70     GameCharacter character) {  
72     NonPlayerCharacter npc =  
73         (NonPlayerCharacter) character;  
75     out.printf("%s: %s\n",  
76         character.getName(), npc.talk());  
77 }
```

CastExamples.java

- ▶ Narrowing Cast von GameCharacter auf NonPlayerCharacter
- ▶ Der Cast **kann** gutgehen
- ▶ ...muss aber nicht
- ▶ Zur Übersetzungszeit **nicht überprüfbar** ob gültig oder nicht

88

# Inhalt

## Vererbung

### Typen in der Hierarchie

„ist-ein“-Beziehung

Widening Cast

Narrowing Cast


ClassCastException

instanceof-Operator

89

## ClassCastException

- ▶ Aufruf von letNPCTalk

```
83  runInvalidNarrowingCastExample  
84 NonPlayerCharacter yennefer = new NonPlayerCharacter(  
85     "Yennefer", "...", 0, 0);  
86 letNPCTalk(yennefer);  
88 Player player = new Player();  
89 letNPCTalk(player);
```

 CastExamples.java

```
Yennefer: Hello, my name is Yennefer! ...  
ClassCastException
```

- ▶ „Player cannot be cast to class NonPlayerCharacter“
- ▶ Erster Aufruf funktioniert
- ▶ Zweiter Aufruf schlägt beim narrowing Cast fehl
- ▶ Java prüft zur **Laufzeit** ob Objekttyp des Parameters mit NonPlayerCharacter kompatibel ist

90

## Vererbung

### Typen in der Hierarchie

„ist-ein“-Beziehung

Widening Cast

Narrowing Cast

ClassCastException

instanceof-Operator

## instanceof-Operator

- ▶ **instanceof**-Operator — siehe auch Kapitel zu **Operatoren**

```
obj instanceof T
```

- ▶ Prüft ob obj in Typ T umgewandelt werden kann
- ▶ **Rückgabewert: boolean**
  - ▶ **true** wenn Objekttyp von obj **allgemeiner** als T
  - ▶ **true** wenn Objekttyp von obj **spezieller** als T
  - ▶ **false** wenn Objekttyp von obj **inkompatibel** zu T
  - ▶ **false** wenn obj == **null**
  - ▶ (Wenn T **interface**: **true** wenn Objekttyp von obj T implementiert; später)
- ▶ Liefert **instanceof true**, erzeugt Cast **garantiert keinen Fehler**

```
if (character instanceof NonPlayerCharacter){  
    // funktioniert garantiert  
    NonPlayerCharacter npc =  
        (NonPlayerCharacter) character;  
    /* ... */  
}
```

## instanceof-Example I

printlnInfo gibt je nach Typ Informationen aus

```
94 public static void printlnInfo(Object obj) {
95     if (obj instanceof Player){
96         Player player = (Player) obj;
97         out.printf("Player: gold=%d\n", player.getGold());
98     } else
99         out.println("Kein Player");
100
101     if (obj instanceof NonPlayerCharacter){
102         NonPlayerCharacter npc = (NonPlayerCharacter) obj;
103         out.printf("NonPlayerCharacter: phrase=%s\n", npc.getPhrase());
104     } else
105         out.println("Kein NonPlayerCharacter");
106
107     if (obj instanceof GameCharacter){
108         GameCharacter character = (GameCharacter) obj;
109         out.printf("GameCharacter: name=%s\n", character.getName());
110     } else
111         out.println("Kein GameCharacter");
112 }
```

93

## instanceof-Example II


```
113     out.println("Kein GameCharacter");
114 }
115 }
```

CastExamples.java

94

## instanceof-Example

- Aufruf von `printInfo` mit `Player`, `NonPlayerCharacter`, [↗ String](#)

```
121  runInstanceof00PExample
122 Player player = new Player();
123 NonPlayerCharacter yennefer = new NonPlayerCharacter(
124     "Yennefer", "...", 0, 0);
125 String s = "Toss a coin to the witcher...";
127 out.println("printInfo(player)");
128 printInfo(player);
129 out.println();
131 out.println("printInfo(yennefer)");
132 printInfo(yennefer);
133 out.println();
135 out.println("printInfo(s)");
136 printInfo(s);
```

 CastExamples.java

95

## instanceof-Example

```
printInfo(player)
Player: gold=0
Kein NonPlayerCharacter
GameCharacter: name=Gerald von Riva

printInfo(yennefer)
Kein Player
NonPlayerCharacter: phrase=...
GameCharacter: name=Yennefer

printInfo(s)
Kein Player
Kein NonPlayerCharacter
Kein GameCharacter
```

96



# Inhalt

## Vererbung

- Überschreiben von Methoden

  - Überschreiben anhand vom Beispiel

  - Signaturen überschriebener Methoden

  - Kovarianz

# Inhalt

## Vererbung

- Überschreiben von Methoden

  - Überschreiben anhand vom Beispiel

  - Signaturen überschriebener Methoden

  - Kovarianz

## Erweiterung des Beispiels

### ► Items: Gegenstände

Item
- name : String
- value : <b>int</b>
+ getName(): String
+ getValue(): <b>int</b>
+ use(player : Player)

► **name**: Name des Gegenstands, z.B. „Schlüssel“

► **value**: Wert des Gegenstands in Gold

► **use**: Benutze Gegenstand

### ► Implementierung `game/Item.java`

### ► Methode use

```
41 public void use(Player player) {  
42     System.out.printf("%s uses %s\n",  
43         player.getName(), name);  
44 }
```

`game/Item.java`

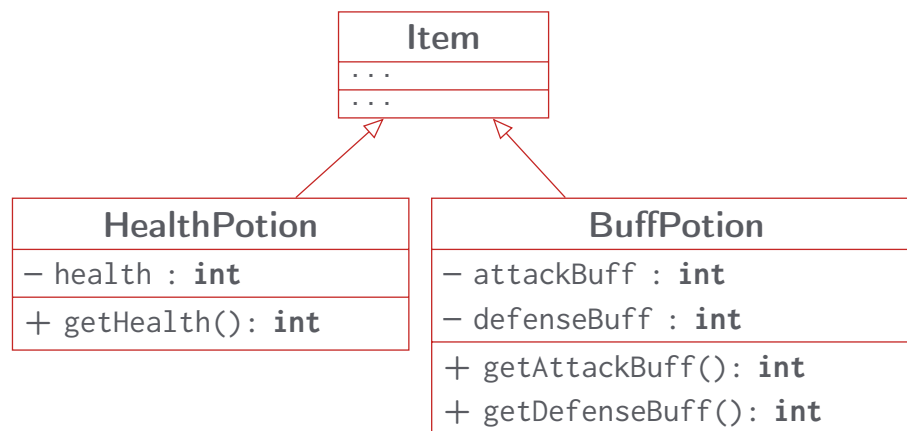
99

## Erweiterung des Beispiels

### ► Zaubertränke

► **HealthPotion**: Heilen des Spieler

► **BuffPotion**: Modifizieren Angriff- und Verteidigungswerte



► **Verhalten** soll sich je nach Trank **unterscheiden**


### ► Methode **use**

► implementiert Verhalten

► wird von Item geerbt

100

## Verhalten bisher I

```
14  runUseItemsExample  
15 var player = new Player();  
17 var key = new Item("Key", 10);  
19 var healthPotion = new HealthPotion(  
20     "Redundant Health Potion of Health", 50, 10);  
22 var buffPotion = new BuffPotion(  
23     "Rage Potion", 100, 10, -5);  
25 key.use(player);  
26 out.println(player);  
28 healthPotion.use(player);  
29 out.println(player);  
31 buffPotion.use(player);  
32 out.println(player);
```

 OverrideExamples.java

101

## Verhalten bisher II

```
Geralt von Riva uses Key  
Player: name="Geralt von Riva", health=100, x=0, y=0, attackPower=1, defensePower=1  
Geralt von Riva uses Redundant Health Potion of Health  
Player: name="Geralt von Riva", health=100, x=0, y=0, attackPower=1, defensePower=1  
Geralt von Riva uses Rage Potion  
Player: name="Geralt von Riva", health=100, x=0, y=0, attackPower=1, defensePower=1
```

102

## Überschreiben der Methode use

- ▶ Bisher keine **Auswirkungen** der Zaubertränke
- ▶ HealthPotion muss use **überschreiben**

```
23 @Override
24 public void use(Player player){
25     player.changeHealth(health);
26 }
```

game/HealthPotion.java

- ▶ **Hinweise**
  - ▶ @Override — Annotation für
    - ▶ **Dokumentation**: „Hier wird was überschrieben!“
    - ▶ **Compiler**: Prüft ob Methode **wirklich überschrieben** wird
  - Optional aber **sehr empfohlen**
  - ▶ Methode use wird nur überschrieben wenn Signatur der von Item.use **genau entspricht** (später mehr)

103

## HealthPotion: Überschreiben der Methode use

Neue Ausgabe mit **unveränderten** Aufrufen von oben

```
Gerald von Riva uses Key
Player: name="Gerald von Riva", health=100, x=0, y=0, attackPower=1, defensePower=1

Player: name="Gerald von Riva", health=110, x=0, y=0, attackPower=1, defensePower=1

Gerald von Riva uses Rage Potion
Player: name="Gerald von Riva", health=110, x=0, y=0, attackPower=1, defensePower=1
```

HealthPotion.use erhöht jetzt tatsächlich die health-Wert

104

## BuffPotion: Überschreiben der Methode use

- ▶ Auch BuffPotion muss use überschreiben...
- ▶ ...hätte aber gerne die Ausgabe von Item.use **beibehalten**
- ▶ **Zur Erinnerung:**
  - ▶ **super()** ruft Konstruktor der **Basisklasse** auf
  - ▶ **this.xyz()** ruft Methode der **eigenen Klasse** auf
- ▶ **super.run(player)** ruft **Implementierung von run** der Basisklasse auf

```
32 @Override
33 public void use(Player player){
34     super.use(player);
35     player.changeAttackPower(attackBuff);
36     player.changeDefensePower(defenseBuff);
37 }
```

game/BuffPotion.java

105

## BuffPotion: Überschreiben der Methode use

Neue Ausgabe mit **unveränderten** Aufrufen von oben

```
Player: name="Geralt von Riva", health=100, x=0, y=0, attackPower=1, ↵
    defensePower=1, #inventory=0
Player: name="Geralt von Riva", health=110, x=0, y=0, attackPower=1, ↵
    defensePower=1, #inventory=0
Geralt von Riva uses Rage Potion
Player: name="Geralt von Riva", health=110, x=0, y=0, attackPower=11, ↵
    defensePower=-4, #inventory=0
```

BuffPotion.use modifiziert jetzt attackPower und defensePower von player

106

## Hinweis zu super-Aufruf

- ▶ Zur Erinnerung: **super**-Konstruktoraufruf muss **erste** Anweisung sein
- ▶ **super**-Methodenaufrufe können
  - ▶ irgendwo stehen
  - ▶ mehrmals vorkommen
- ▶ Beispiel

```
@Override
public void use(Player player){
    super.use(player);
    player.changeAttackPower(attackBuff);
    super.use(player);
    player.changeDefensePower(defenseBuff);
    super.use(player);
}
```

- ▶ Normalerweise: nur **einmal** und eher am **Anfang**

## Inhalt

### Vererbung

#### Überschreiben von Methoden

Überschreiben anhand vom Beispiel

Signaturen überschriebener Methoden

Kovarianz

## Parameterlisten überschriebener Methoden

- ▶ Signatur der **überschreibenden** und **überschriebenen** müssen übereinstimmen
- ▶ Alternative Implementierung von use in HealthPotion

```
@Override
public void use() { // FEHLER
    System.out.println("Pouring potion on the floor");
}
```

„The method use() must override a supertype method“

- ▶ Selbst wenn die Signatur **kompatibel** wäre

```
@Override
public void use(GameCharacter character) { // FEHLER
    character.changeHealth(health);
}
```

- ▶ **Hinweis:** Fehlermeldung wird nur mit @Override angezeigt

109

## Sichtbarkeit überschriebener Methoden

- ▶ Sichtbarkeit darf beim Überschreiben nur **größer** werden
- ▶ **Zur Erinnerung:** **private** < Paket < **protected** < **public**
- ▶ Beispiel

```
@Override
void use(Player player){ // FEHLER
    player.changeHealth(health);
}
```

„Cannot reduce the visibility of inherited method“

- ▶ Vergrößerung der Sichtbarkeit **möglich**
  - ▶ **protected** Item.use(Player) → **public** HealthPotion.use(Player)
  - ▶ Item.use(Player) → **protected/public** HealthPotion.use(Player)
  - ▶ **private** Item.use(Player) → **kein Überschreiben möglich**

110

## Sichtbarkeit überschriebener Methoden

- ▶ Warum kann die Sichtbarkeit nur **größer** werden?
- ▶ Grund: Substitutionsprinzip
  - ▶ Überall wo Item **verwendet werden kann...**
  - ▶ **muss** auch auch HealthPotion **verwendet werden können**
- ▶ Angewandt auf **public** Item.use(Player)
  - ▶ **Annahme:** **protected** HealthPotion.use(Player)
  - ▶ **Betrachte**

```
public void useOnPlayer(Player player, Item item){  
    item.use(player);  
}
```

```
var key = new Item(...);  
var healthPotion = new HealthPotion(...);  
useOnPlayer(player, key); // OK, da Item  
useOnPlayer(player, healthPotion); // FEHLER
```

- ▶ Widerspruch zum **Substitutionsprinzip**

111

## Warum @Override wichtig ist

- ▶ Gibt de...oopbasics.shapes.Point2D auf Konsole aus

```
4 import de.hawlandshut.java1.oopbasics.shapes.Point2D;  
5 public class Point2DPrinter {  
6     public void printPoint(Point2D p){  
7         System.out.printf("Super-Class: x=%d, y=%d",  
8             p.getX(), p.getY());  
9     }  
10 }
```

Point2DPrinter.java

- ▶ Erweiterung mit „überschriebener“ Methode

```
4 import java.awt.geom.Point2D;  
5 public class SubPoint2DPrinter extends Point2DPrinter {  
6     public void printPoint(Point2D p){  
7         System.out.printf("Sub-Class: x=%f, y=%f",  
8             p.getX(), p.getY());  
9     }  
10 }
```


SubPoint2DPrinter.java

112



## Warum @Override wichtig ist

### ► Test

```
38  runForgottenOverrideMessExample  
39 var printer = new SubPoint2DPrinter();  
40 var p = new de.hawlandshut.java1.oopbasics.shapes.Point2D(1,2);  
42 printer.printPoint(p);
```

 OverrideExamples.java

### ► Ergebnis

Super-Class: x=1, y=2

- Es wird **immer noch** Methode der Superklasse aufgerufen
- Grund: **unterschiedliche Parameter**
  - Point2DPrinter.printPoint(de...shapes.Point2D)
  - SubPoint2DPrinter.printPoint(java.awt.geom.Point2D)

113

## Warum @Override wichtig ist

### ► Mit @Override wäre das **nicht passiert**

```
import java.awt.geom.Point2D;  
public class SubPoint2DPrinter extends Point2DPrinter {  
    @Override  
    public void printPoint(Point2D p){ // FEHLER  
        System.out.printf("Sub-Class: x=%d, y=%d",  
            p.getX(), p.getY());  
    }  
}
```

„The method printPoint(Point2D) must override supertype method“

- **Also:** @Override verwenden!

114

# Inhalt

## Vererbung

### Überschreiben von Methoden

Überschreiben anhand vom Beispiel

Signaturen überschriebener Methoden

Kovarianz

115

## Kovarianz

- ▶ Rückgabewerte können beim Überschreiben **spezieller** werden
- ▶ `Item.doublePotentVersion` erstellt **doppelt so starke** (und **teurere**) Version

```
48 public Item doublePotentVersion() {  
49     return new Item("Powerful " + getName(),  
50         2*value);  
51 }
```

game/Item.java

- ▶ `HealthPotion.doublePotentVersion` überschreibt und hat **spezielleren** Rückgabotyp


```
36 @Override  
37 public HealthPotion doublePotentVersion() {  
38     return new HealthPotion(  
39         "Even healthier " + getName(),  
40         2*getValue(), 2*health);  
41 }
```

game/HealthPotion.java

116

# Kovarianz

## ► Aufruf

```
51  runCovarianceExample  
52 Item expensiveKey = key.doublePotentVersion();  
54 // kein Cast notwendig:  
55 HealthPotion powerfulHealthPotion =  
56     healthPotion.doublePotentVersion();  
58 out.println(expensiveKey);  
59 out.println(powerfulHealthPotion);
```

 OverrideExamples.java

## ► Praktisch: kein narrowing Cast notwendig

## ► Ergebnis

```
Item: name = "Powerful Key", value = 20  
HealthPotion: name = "Even healthier Redundant Health Potion of Health", value ↔  
    = 100, HealthPotion: health = 20
```

117

# Warum funktioniert Kovarianz?

## ► Warum funktioniert Kovarianz?

## ► Substitutionsprinzip

- Dort wo `Item.doublePotentVersion()` verwendet werden kann...
- muss auch `HealthPotion.doublePotentVersion` verwendet werden können
- Das gilt: Rückgabetyt von `HealthPotion.doublePotentVersion` ist spezieller

## ► Gleicher Grund: Rückgabetyt darf nicht allgemeiner werden

118

# Inhalt

## Vererbung

### Überschreiben der Methoden von Object

`Object.toString()`

`Object.equals()` und `Object.hashCode()`

`Object.clone()`

Zusammenfassung: Überschreiben der Methoden von Object

## Überschreiben der Methoden von Object

- ▶ Zur Erinnerung: Jede Klasse erbt von `Object`
- ▶ Manche der Methoden können (sollten) überschrieben werden
  - ▶ `String toString()` — liefert `String`-Repräsentation
  - ▶ `boolean equals(Object obj)` — Vergleich mit anderem Objekt (Wertgleichheit)
  - ▶ `int hashCode()` — Berechnung eines Hashwerts
  - ▶ `Object clone()` — Kopieren des Objekts

# Inhalt

## Vererbung

### Überschreiben der Methoden von Object

`Object.toString()`

`Object.equals()` und `Object.hashCode()`

`Object.clone()`

Zusammenfassung: Überschreiben der Methoden von Object

121

## Object.toString()

- ▶ **Dokumentation** von `toString()`: „Returns a string representation of the object. In general, the `toString` method returns a string that 'textually represents' this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.“
- ▶ Implementierung in [↗ Object](#) liefert für Player

```
de.hawlandshut.java1.oop.game.Player@421faab1
```

Klassenname@Hashwert als Hexadezimalzahl

- ▶ **Überschreiben**
  - ▶ **Wie?** Das ist einem selbst überlassen
  - ▶ Ausgabe für **Debugging**
  - ▶ **Nicht** unbedingt für Nutzerausgaben

122

## GameCharacter.toString()

- ▶ Vorschlag für Implementierung in GameCharacter

```
100 @Override
101 public String toString() {
102     return String.format(
103         "%s: name=\"%s\", health=%d, x=%d, y=%d",
104         getClass().getSimpleName(), name, health, x, y);
105 }
```

game/GameCharacter.java

- ▶ [String.format](#) nutzen für Lesbarkeit
- ▶ Informationen
  - ▶ **Objektyp**: `getClass().getSimpleName()` (alternativ `getName()` für voll qualifizierten Namen)
  - ▶ **Attribute** der Klasse — **ohne Attribute der Basisklasse** (s. unten)

123

## NonPlayerCharacter.toString()

- ▶ NonPlayerCharacter.toString()

```
50 @Override
51 public String toString() {
52     return String.format("%s, phrase=%s",
53         super.toString(), phrase);
54 }
```


game/NonPlayerCharacter.java

- ▶ **Format**
  - ▶ `super.toString()` — Informationen der Basisklasse
  - ▶ `phrase` — Zusätzliche Attribute der Ableitung
- ▶ **Vorteile**
  - ▶ `toString` nutzt Implementierung der Basisklasse
  - ▶ keine Wiederholung von Code

124

## NonPlayerCharacter.toString()

### ► Test

```
12  runOverrideToStringExample  
13 NonPlayerCharacter yennefer = new NonPlayerCharacter(  
14     "Yennefer", "...", 0, 0);  
15 System.out.println(yennefer);
```

 ObjectOverrideExamples.java

```
NonPlayerCharacter: name="Yennefer", health=1, x=0, y=0, phrase=...
```

125

## Player.toString()

### ► Gleiche Idee bei Player.toString()

```
161 @Override  
162 public String toString() {  
163     return String.format(  
164         "%s, attackPower=%d, defensePower=%d, #inventory=%d",  
165         super.toString(), attackPower,  
166         defensePower, inventory != null ? inventory.size() : 0);  
167 }
```

 game/Player.java

### ► Ausgabe

```
Player: name="Geralt von Riva", health=100,  
x=0, y=0, attackPower=1, defensePower=1, #inventory=0
```

126

# Inhalt

## Vererbung

### Überschreiben der Methoden von Object

`Object.toString()`

`Object.equals()` und `Object.hashCode()`

`Object.clone()`

Zusammenfassung: Überschreiben der Methoden von Object

## Object.equals()

- ▶ `boolean Object.equals(Object obj)`
  - ▶ Prüft auf **Gleichheit**
  - ▶ Implementierung in `Object` prüft nur **Identität**
  - ▶ **Überschreiben**: Siehe Kapitel zu Vergleich von Objekten
- ▶ **Achtung**: Wertevergleich muss. . .
  - ▶ Attribute der **Klasse selbst** vergleichen
  - ▶ Attribute der **Basisklasse** (und **darüber**) vergleichen
- ▶ Von IDE generierte `equals`-Methode **prüft nur Attribute** der Klasse!



## IDE-Version von NonPlayerCharacter.equals()

- ▶ Von IDE generierte NonPlayerCharacter.equals()


```
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null) return false;
    if (getClass() != obj.getClass()) return false;
    NonPlayerCharacter other = (NonPlayerCharacter) obj;
    if (!Objects.equals(phrase, other.phrase))
        return false;
    return true;
}
```

- ▶ Problem:
  - ▶ Prüft nur Gleichheit von phrase
  - ▶ **Ignoriert** Attribute von GameCharacter

129

## IDE-Version von NonPlayerCharacter.equals()

- ▶ Test

```
21  runWrongEqualsExample
22 NonPlayerCharacter yennefer =
23     new NonPlayerCharacter("Yennefer", "*sigh*", 0, 0);
24 NonPlayerCharacter jaskier =
25     new NonPlayerCharacter("Jaskier", "*sigh*", 2, 6);
27 System.out.printf("yennefer.equals(yaskier): %b\n",
28     yennefer.equals(jaskier));
```

 ObjectOverrideExamples.java

```
yennefer.equals(yaskier): true
```

- ▶ phrase stimmt überein — sonst aber gar nichts

130

## GameCharacter.equals()

### ► Idee

- Wir implementieren GameCharacter.equals()...
- und nutzen diese in NonPlayerCharacter.equals()

### ► GameCharacter.equals() von IDE generiert (hier OK!)

```
120 @Override
121 public boolean equals(Object obj) {
122     if (this == obj) return true;
123     if (obj == null) return false;
124     if (getClass() != obj.getClass()) return false;
126     GameCharacter other = (GameCharacter) obj;
127     if (health != other.health) return false;
128     if (!Objects.equals(name, other.name)) return false;
129     if (x != other.x) return false;
130     if (y != other.y) return false;
132     return true;
133 }
```

game/GameCharacter.java

131

## Korrekte NonPlayerCharacter.equals()

### ► Nun können wir GameCharacter.equals() in NonPlayerCharacter.equals() nutzen


```
72 @Override
73 public boolean equals(Object obj) {
74     // Jetzt optional
75     // if (this == obj) return true;
76     // if (obj == null) return false;
77     // if (getClass() != obj.getClass()) return false;
79     if (!super.equals(obj)) // NEU
80         return false;
82     NonPlayerCharacter other = (NonPlayerCharacter) obj;
83     if (!Objects.equals(phrase, other.phrase))
84         return false;
86     return true;
87 }
```

game/NonPlayerCharacter.java

132

## Korrekte `NonPlayerCharacter.equals()` — Test

- ▶ Jetzt funktioniert's



```
34  runRightEqualsExample
35 NonPlayerCharacter yennefer =
36     new NonPlayerCharacter("Yennefer", "*sigh*", 0, 0);
37 NonPlayerCharacter yenneferClone =
38     new NonPlayerCharacter(yennefer);
39 NonPlayerCharacter jaskier =
40     new NonPlayerCharacter("Jaskier", "*sigh*", 2, 6);
42 System.out.printf("yennefer.equals(yaskier): %b\n",
43     yennefer.equals(jaskier));
44 System.out.printf("yennefer.equals(yenneferClone): %b\n",
45     yennefer.equals(yenneferClone));
```

 ObjectOverrideExamples.java

```
yennefer.equals(yaskier): false
yennefer.equals(yenneferClone): true
```

133

## `Object.hashCode()`

- ▶ **Hashwert**: Abbildung der **relevanten** Objektattribute auf **int**-Wert
  - ▶ Für Hashwert-basierte Datenstrukturen (z.B.  **HashMap**)
  - ▶ Für schnellen Vergleich
    - ▶ `x.hashCode() != y.hashCode()` — x und y garantiert **ungleich**
    - ▶ `x.hashCode() == y.hashCode()` — x und y **eventuell gleich**, weiterer Test mit `x.equals(y)`
  - ▶ Gute Hashwert-Generierung ist **eine Kunst für sich**
  - ▶  **Object.hashCode()** generiert Hashwert mit Speicheradresse
    - ▶ Nicht ganz schlecht
    - ▶ Aber auch **nicht sehr gut**

134

## IDE-Version von NonPlayerCharacter.hashCode()

- ▶ IDE generiert NonPlayerCharacter.hashCode()


```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((phrase == null) ? 0 : phrase.hashCode());
    return result;
}
```

- ▶ Idee
  - ▶ Fange mit result = 1 an
  - ▶ Multipliziere **Hashwert des Attributs** mit Primzahl und addiere zu result
  - ▶ Fahre mit nächstem Attribut fort
- ▶ Situation ähnlich wie bei generierter Version von equals
  - ▶ Bezieht phrase mit ein
  - ▶ **Ignoziert** aber Attribute der Basisklasse GameCharacter

135

## IDE-Version von NonPlayerCharacter.hashCode() — Test

- ▶ Test

```
51  runAddHashCodeExample
52 NonPlayerCharacter yennefer =
53     new NonPlayerCharacter("Yennefer", "*sigh*", 0, 0);
54 NonPlayerCharacter jaskier =
55     new NonPlayerCharacter("Jaskier", "*sigh*", 2, 6);
57 System.out.printf("yennefer.hashCode(): %d%n",
58     yennefer.hashCode());
59 System.out.printf("jaskier.hashCode(): %d%n",
60     jaskier.hashCode());
```

 ObjectOverrideExamples.java

```
yennefer.hashCode(): 1311859592
jaskier.hashCode(): 1311859592
```

- ▶ Schlechte Hashwerte
  - ▶ Objekt sind **sehr unterschiedlich...**
  - ▶ liefern aber **gleichen Hashwert**

136

## GameCharacter.hashCode()

- ▶ Idee wie oben: Wir implementieren GameCharacter.hashCode()...
- ▶ und nutzen diese in NonPlayerCharacter.hashCode()
- ▶ Siehe `game/GameCharacter.java` für Implementierung (IDE-generiert)
- ▶ Modifizierte Version von NonPlayerCharacter.hashCode()

```
62 @Override
63 public int hashCode() {
64     final int prime = 31;
65     int result = super.hashCode(); // NEU
66     result = prime * result + ((phrase == null) ? 0 : phrase.hashCode());
67     return result;
68 }
```


`game/NonPlayerCharacter.java`

- ▶ Hinweis: Wir starten nicht mit Hashwert 1 sondern mit Hashwert der Basisklasse

137

## IDE-Version von NonPlayerCharacter.hashCode() — Test

- ▶ Test

```
67  runGoodHashCodeExample
68 NonPlayerCharacter yennefer =
69     new NonPlayerCharacter("Yennefer", "*sigh*", 0, 0);
70 NonPlayerCharacter yenneferClone =
71     new NonPlayerCharacter(yennefer);
72 NonPlayerCharacter jaskier =
73     new NonPlayerCharacter("Jaskier", "*sigh*", 2, 6);
75 System.out.printf("yennefer.hashCode(): %d\n",
76     yennefer.hashCode());
77 System.out.printf("yenneferClone.hashCode(): %d\n",
78     yenneferClone.hashCode());
79 System.out.printf("jaskier.hashCode(): %d\n",
80     jaskier.hashCode());
```

`ObjectOverrideExamples.java`

138

### ► Test

```
yennefer.hashCode(): 1598020687  
yenneferClone.hashCode(): 1598020687  
jaskier.hashCode(): -1704002594
```

### ► Gute Hashwerte

- Unterschiedlich für unterschiedliche Instanzen
- Gleich für gleiche Instanzen

## Inhalt

### Vererbung

#### Überschreiben der Methoden von Object

Object.toString()

Object.equals() und Object.hashCode()

Object.clone()

Zusammenfassung: Überschreiben der Methoden von Object

## Kopieren von Objekten über Kopier-Konstruktor

- ▶ Kopieren von Objekten bisher: **Kopier-Konstruktor**
- ▶ **Siehe auch** Kapitel „Kopieren von Objekten“
- ▶ **Kopier-Konstrukturen** haben ein **Problem**
  - ▶ Sie funktionieren nur wirklich für Klassen, die von [Object](#) ableiten
  - ▶ **Grund**: Konstruktoren sind **statisch gebunden**
  - ▶ **Beispiel**: Merchant soll einen (**flachen**) Kopier-Konstruktor bekommen

```
18 public Merchant(Merchant other){
19     super(other);
20     this.stock = other.getStock().clone();
21 }
```

game/Merchant.java

- ✓ Ruft **Kopier-Konstruktor** von NonPlayerCharacter auf
- ✓ Kopiert zusätzliche Attribute (flach: **Wertzuweisung**)

141

## Kopieren von Objekten über Kopier-Konstruktor

- ▶ Kopier-Konstruktor von NonPlayerCharacter funktioniert genauso

```
22 public NonPlayerCharacter(NonPlayerCharacter other){
23     super(other);
24     this.phrase = other.getPhrase();
25 }
```

game/NonPlayerCharacter.java

- ▶ Kopier-Konstruktor von GameCharacter kopiert die Attribute

```
37 public GameCharacter(GameCharacter gameCharacter){
38     this.name = gameCharacter.getName();
39     this.health = gameCharacter.getHealth();
40     this.x = gameCharacter.getX();
41     this.y = gameCharacter.getY();
42 }
```

game/GameCharacter.java

- ▶ **Bisher**: Alles gut, oder?

142


## Kopieren von Objekten über Kopier-Konstruktor

- ▶ cloneNPC „klont“ NonPlayerCharacter

```
85 public static NonPlayerCharacter
86     cloneNPC(NonPlayerCharacter npc) {
87     return new NonPlayerCharacter(npc);
88 }
```

ObjectOverrideExamples.java

- ▶ Aufruf

```
94  runCopyConstructorGoneWrongExample
95 var merchant = new Merchant("Bram", stock, 10, 5);
96 out.println(merchant);
98 var merchantClone = new Merchant(merchant);
99 out.println(merchantClone);
101 var anotherClone = cloneNPC(merchant);
102 out.println(anotherClone);
```

ObjectOverrideExamples.java

143


## Kopieren von Objekten über Kopier-Konstruktor

- ▶ Ergebnis

```
Merchant: name="Poor Merchant", health=1, x=10, y=5, ...
Merchant: name="Poor Merchant", health=1, x=10, y=5, ...
NonPlayerCharacter: name="Poor Merchant", health=1, ...
```

- ▶ Ergebnis von cloneNPC ist **nicht** vom Typ Merchant
- ▶ Grund:

```
new NonPlayerCharacter(npc);
```

- ▶ Erzeugt NonPlayerCharacter
- ▶ Grund: Konstruktor-Aufruf kann **nicht** dynamisch an Objekttyp von npc gebunden werden
- ▶ Kopier-Konstruktor geht nur wenn Objekttyp zur Übersetzung bekannt
- ▶ Aber: Methoden sind **dynamisch** gebunden
- ▶ Enter  `Object.clone()`

144



## Object.clone()

- ▶ ↗ Object.clone zur Erstellung von **flachen** Kopien
- ▶ Vollständige Signatur in ↗ Object

```
protected Object clone()  
    throws CloneNotSupportedException
```

- ▶ Problem(e)
  - ▶ ↗ Object.clone ist **protected**, kein Aufruf von **außen**
  - ▶ Wir können ↗ Object.clone in z.B. Player
    - ▶ überschreiben
    - ▶ **public** machen
    - ▶ über **Kovarianz** den Rückgabewert **spezialisieren**

```
171 @Override public Player clone()  
172     throws CloneNotSupportedException{  
173     return (Player) super.clone();  
174 }
```

game/Player.java

Aber **super.clone()** führt zu ↗ CloneNotSupportedException

- ▶ Verschiebung auf Diskussion von **interface** Cloneable

145

## Inhalt

### Vererbung

#### Überschreiben der Methoden von Object

Object.toString()

Object.equals() und Object.hashCode()

Object.clone()

Zusammenfassung: Überschreiben der Methoden von Object

146

## Zusammenfassung

Methode	Hinweise
toString()	IDE oder selbst, evtl. Basisklasse einbeziehen
equals()	IDE-Version unbedingt mit <b>super.equals()</b> erweitern!
hashCode()	IDE-Version unbedingt mit <b>super.hashCode()</b> erweitern!
clone()	später ( <b>interface</b> Cloneable)

147

## Inhalt

### Vererbung

Finale Methoden und Klassen

148

## Finale Klassen

- ▶ Ableiten von Klassen kann **verhindert** werden

```
7 public final class HealthPotion extends Item {
```

game/HealthPotion.java

- ▶ **Modifizierer final** verhindert Ableitung

```
public class SpecialHealthPotion  
    extends HealthPotion // FEHLER
```

„Cannot subclass the final class HealthPotion“

- ▶ **Gründe für final**
  - ▶ Verhinderung von **Modifikationen**
  - ▶ **Sinnhaftigkeit** z.B. bei java.lang.Math
  - ▶ Dokumentation: Diese Klasse **soll** nicht verändert werden

149

## Finale Methoden

- ▶ Überschreiben von Methoden kann ebenfalls **verhindert** werden
- ▶ **Beispiel:** GameCharacter.move(int,int) bewegt Character

```
93 protected final void move(int dx, int dy) {  
94     x += dx;  
95     y += dy;  
96 }
```

game/GameCharacter.java

- ▶ Das Verhalten vom move soll **nicht verändert** werden
- ▶ **Beispiel:** Player

```
@Override  
protected void move(int dx, int dy){ // FEHLER  
    x -= dx; // mirror-inverted  
    y -= dy;  
}
```

„Cannot override move in Player“

- ▶ **Gründe** wie bei **final** Klassen

150

# Inhalt

## Vererbung

Dynamische und statische Bindung

151

## Dynamische Bindung

### ► Zur Erinnerung

```
Item item = new HealthPotion(...);
```

item : Item

HealthPotion : name=...

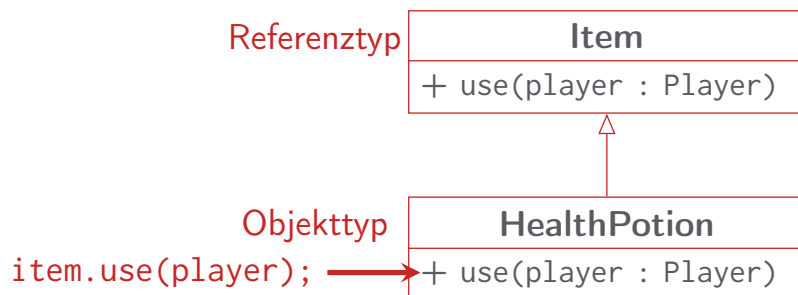
- Item: Referenztyp (von item)
- HealthPotion: Objekttyp

152

# Dynamische Bindung

- ▶ **Dynamische Bindung**
  - ▶ Bei **Methodenaufrufen** überschreibbarer Methoden
  - ▶ entscheidet die JVM **zur Laufzeit**
  - ▶ anhand des **Objektyps**
  - ▶ **welche Methode** aufgerufen wird
- ▶ **Beispiel**

```
Item item = new HealthPotion(...);  
item.use(player);
```



153

## Dynamische Bindung: Beispiel

- ▶ Item mit Implementierung von use

```
41 public void use(Player player) {  
42     System.out.printf("%s uses %s\n",  
43         player.getName(), name);  
44 }
```

game/Item.java

- ▶ HealthPotion überschreibt use

```
23 @Override  
24 public void use(Player player){  
25     player.changeHealth(health);  
26 }
```

game/HealthPotion.java

154

## Dynamische Bindung: Beispiel

### ► Aufruf

```
Item item = new HealthPotion("Health Potion", 10);  
item.use(player);
```

- Referenztyp: Item
- Objekttyp: HealthPotion

### ► **Dynamische Bindung:** Objekttyp bestimmt aufgerufene Methode HealthItem.use

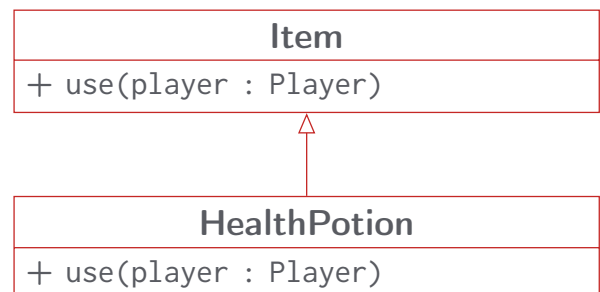
### ► Ergebnis

```
Player: ..., health=100, ...  
Player: ..., health=110, ...
```

155

## Dynamische Bindung

- Dynamische Bindung, auch
  - **Late Binding**, späte Bindung
  - **Polymorphie** im Kontext objektorientierter Programmierung
- Eigentliches Verhalten wird erst zur **Laufzeit** bestimmt
- **Vorteil**
  - **Einfachheit:** Arbeiten mit **abstrakter Schnittstelle**, statt vieler unterschiedlicher Typen
  - **Erweiterbarkeit:** Neues Verhalten kann später hinzugefügt werden
- **Nachteil**
  - **Erhöhte Laufzeit:** Prüfung Objekttyp statt „einfacher“ Sprung in Methode



156

## Statische Bindung

- ▶ **Nicht jeder** Methodenaufruf ist dynamisch gebunden
  - ▶ **private**-Methoden — nicht sichtbar/überschreibbar
  - ▶ **final**-Methoden — nicht überschreibbar
  - ▶ **static**-Methoden — kein Objekttyp
- ▶ In diesen Fällen **statische Bindung**
  - ▶ Java bestimmt zur **Übersetzungszeit** welche Methode aufgerufen wird
  - ▶ **Early Binding**, frühe Bindung
- ▶ **Beispiele**
  - ▶ Statische Methode:

`Math.sin(3.1415);`
  - ▶ **final** `GameCharacter.move(int,int)`

157

## Überdeckung vs. Überschreibung

- ▶ Betrachte

```
4 public class KeepItPrivate{
5     private String message(){
6         return "This is private!";
7     }
9     public void saySomething(){
10        System.out.println(message());
11    }
12 }
```

`KeepItPrivate.java`

```
4 public class MakeItPublic extends KeepItPrivate{
5     public String message(){
6         return "I make it public!";
7     }
8 }
```

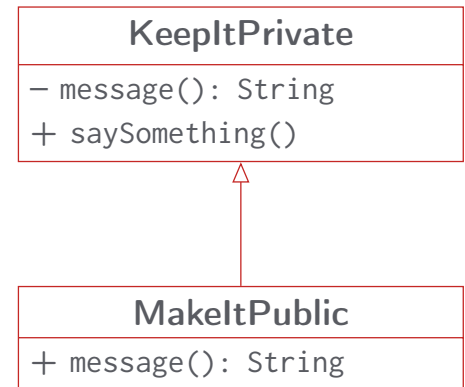
`MakeItPublic.java`

158

## Überdeckung vs. Überschreibung

- ▶ KeepItPrivate.message() ist **private**
- ▶ **public** MakeItPublic.message()
  - ▶ überschreibt KeepItPrivate.message() **nicht**
  - ▶ **überdeckt** KeepItPrivate.message()
- ▶ **private**-Methoden sind nicht **sichtbar** in Unterklassen
- ▶ **Hinweis:** @Override würde zu Fehler führen

```
@Override
public String message(){ // FEHLER
    return "I make it public!";
}
```



159

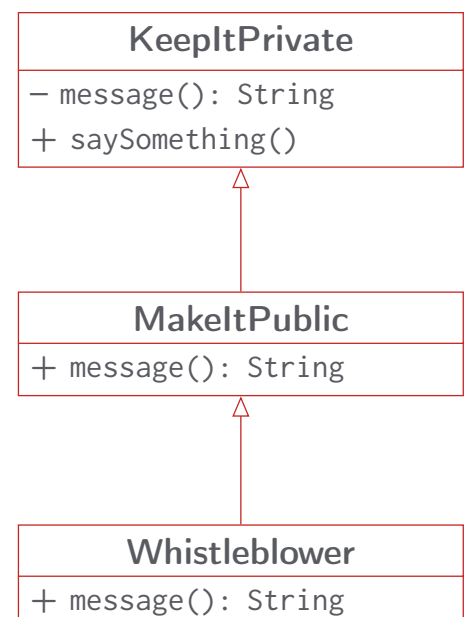
## Überdeckung vs. Überschreibung

- ▶ Noch eine Klasse

```
4 public class WhistleBlower extends MakeItPublic {
5     @Override
6     public String message(){
7         return "There are no waffles of mass destruction!";
8     }
9 }
```

WhistleBlower.java

- ▶ Überschreibt MakeItPublic.message()




160



## Überdeckung vs. Überschreibung

### ► Aufruf

```
65  runDynamicVsStaticBindingExample
66 var priv = new KeepItPrivate();
67 var pub = new MakeItPublic();
68 var whistleblower = new WhistleBlower();
70 priv.saySomething();
71 pub.saySomething();
72 whistleblower.saySomething();
```

 OverrideExamples.java

### ► Was ist die Ausgabe und warum?

### ► Ausgabe

```
This is private!
This is private!
This is private!
```

161

## Überdeckung vs. Überschreibung

### ► Begründung:

- KeepItPrivate.saySomething() ruft die **private**-Methode KeepItPrivate.message() auf
  - Die ist **statisch gebunden**
  - Hat **nichts** mit MakeItPublic.message() zu tun
- MakeItPublic müsste saySomething **überschreiben** um **dynamisch gebundene** Methode MakeItPublic.message() aufzurufen

```
@Override
public void saySomething() {
    System.out.println(message());
}
```

```
This is private!
I make it public!
There are no waffles of mass destruction!
```

162

# Inhalt

## Vererbung

### Abstrakte Klassen und Methoden

163

## GameCharacter.update()

- ▶ GameCharacter: Oberklasse aller Spiel-Characters
- ▶ **Neue Methode** update()
  - ▶ Implementiert **Verhalten**: z.B. Bewegung, Angriff
  - ▶ **Muss** von Unterklassen implementiert werden

GameCharacter
...
+ update()
...

- ▶ **Erster Ansatz**: GameCharacter

```
public void update(){  
}
```

- ▶ **Leere** Implementierung (unschön)
- ▶ Erzwingt **kein** Überschreiben

164

## Abstrakte Klassen und Methoden

- Bessere Lösung: Abstrakte Methode GameCharacter

```
51 public abstract void update();
```

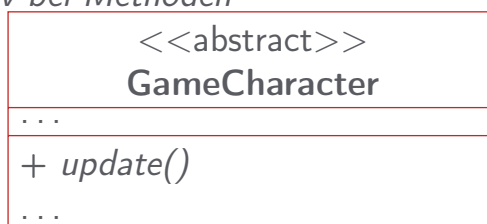
game/GameCharacter.java

- Abstrakte Methoden
  - Modifizierer **abstract**
  - Nur **Signatur**
  - Keine **Implementierung**
- Klassen mit abstrakten Methoden, **müssen** selbst **abstrakt** sein

```
9 public abstract class GameCharacter
```

game/GameCharacter.java

- UML: «abstract» und *kursiv bei Methoden*



165

## Abstrakte Klassen und Methoden

- Was sind die **Konsequenzen**?
  - kein **new** auf GameCharacter möglich

```
GameCharacter character =  
    new GameCharacter("Abstract Ghost", 1000, 0, 0); // FEHLER
```

„GameCharacter is abstract; cannot be instantiated“

- **Nicht-abstrakte Subklassen** müssen abstrakte Methoden implementieren; z.B. NonPlayerCharacter

```
41 public void update() {  
42     System.out.println(talk());  
43     int dx = (int) Math.round(2*Math.random()-1);  
44     int dy = (int) Math.round(2*Math.random()-1);  
45     move(dx,dy);  
46 }
```

game/NonPlayerCharacter.java

166

## Abstrakte Klassen und Methoden

### ► Was sind die Konsequenzen?

- Implementiert eine Subklasse eine abstrakte Methode **nicht**, so ist sie selber abstrakt
- **Beispiel** Enemy: update taucht nicht auf

```
4 public abstract class Enemy extends GameCharacter
5 {
6     private final int attackPower;
7
8     public Enemy(String name, int x, int y, int health, int attackPower) {
9         super(name, health, x, y);
10        this.attackPower = attackPower;
11    }
12
13    public int getAttackPower() {
14        return attackPower;
15    }
```

game/Enemy.java

167

## Regeln für abstrakte Klassen

- Implementierte Methoden können in Subklassen **nicht mehr abstrakt** werden

```
public abstract class GhostCharacter extends NonPlayerCharacter{
    public abstract void update(); // FEHLER
}
```

- Konstruktoren können **nicht** abstrakt sein
- Abstrakte Klassen können **nicht-abstrakte** Methoden enthalten (**partiell abstrakte Klassen**); z.B. GameCharacter.move(int,int)

```
93 protected final void move(int dx, int dy) {
94     x += dx;
95     y += dy;
96 }
```

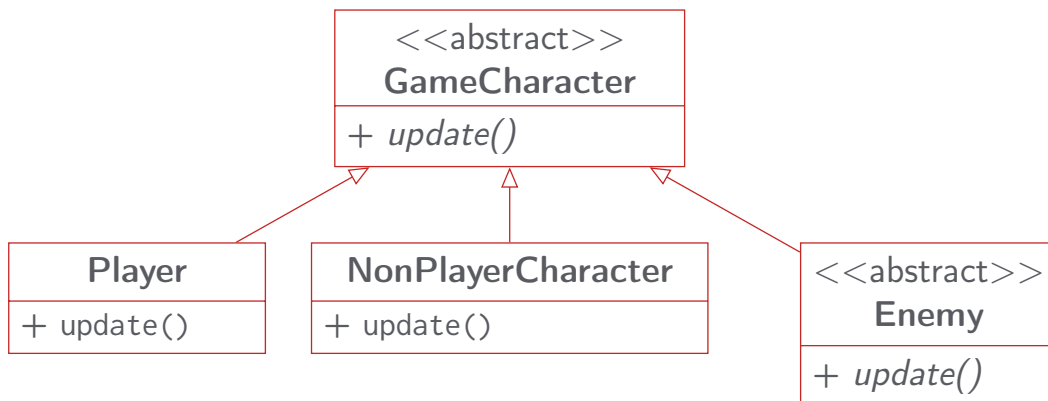
game/GameCharacter.java

- Abstrakte Klassen können **nur abstrakte** Methoden beinhalten (**Pure abstrakte Klassen**)

168

## Warum abstrakte Klassen?

- ▶ Abstrakte Klassen definieren **gemeinsame Schnittstelle und (partiell) Verhalten** einer Kategorie von Objekten
  - ▶ `GameCharacter` definiert `health`, `Position`, etc.
- ▶ **Trennung** von Schnittstelle (+ ein wenig Implementierung) und konkreter Implementierung in Subklassen
  - ▶ `GameCharacter.move(int, int)` definiert wie eine Bewegung ausgeführt wird
  - ▶ `GameCharacter.update()` lässt Verhalten für Subklassen offen



169

## Beispiel

- ▶ `NonPlayerCharacter.update()`: redet, läuft in zufällige Richtung

```
41 public void update() {
42     System.out.println(talk());
43     int dx = (int) Math.round(2*Math.random()-1);
44     int dy = (int) Math.round(2*Math.random()-1);
45     move(dx,dy);
46 }
```

game/NonPlayerCharacter.java

- ▶ `Player.update()`: wartet auf Nutzereingabe

```
99 public void update() {
100     out.println("Wohin? (wasd)");
101     // ...

```

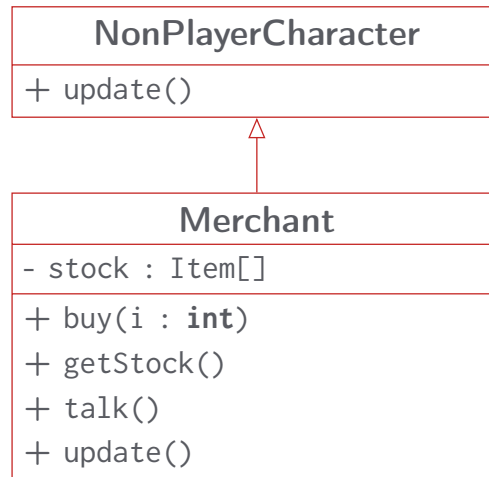
game/Player.java

- ▶ `Enemy.update()`: abstrakt, je nach Gegnertyp anders

170

## Klasse Merchant

- ▶ Neue Klasse Merchant: Händler
  - ▶ stock — Array an Items zum Verkauf
  - ▶ update() — Bleibt wo er ist
  - ▶ talk() — Grüßt und listet Waren auf
  - ▶ buy(i : int) — Kaufen eines Items



- ▶ Implementierung `game/Merchant.java`

171

## Klasse Merchant

- ▶ `Merchant.update()`

```
54 @Override
55 public void update(){
56     out.println(talk());
57 }
```

`game/Merchant.java`

- ▶ `Merchant.buy()`

```
30 public Item buy(int index){
31     Item item = stock[index];
32     stock[index] = null;
33     return item;
34 }
```

`game/Merchant.java`

172

## Klasse Merchant

### ► Merchant.talk()

```
38 @Override
39 public String talk(){
40     StringBuilder builder = new StringBuilder();
41     builder.append(super.talk());
42
43     builder.append("\nMy stock: \n");
44     for (var item : stock){
45         builder.append(" " + item + "\n");
46     }
47
48     return builder.toString();
49 }
```

game/Merchant.java

173

## Game Loop

### ► Jetzt können wir die „Game Loop“ implementieren


### ► Zuerst das einfache Setup

- Siehe GameCharacter[] GameTest.setup() in game/GameTest.java
- Player — Geralt von Riva
- NonPlayerCharacter — Yennefer
- NonPlayerCharacter — Jaskier der Barde
- Merchant — Händler mit drei Items
- keine Gegner

174

## Game Loop

### ► Game Loop

```
42  runGameLoop
43 GameCharacter[] characters = gameSetup();
44 Player player = (Player) characters[0];
46 while (player.isAlive()) {
47     for (GameCharacter character : characters){
48         character.update();
49         out.println(character);
50         out.println();
51     }
52 }
```

 game/GameTest.java

- Game Loop arbeitet mit **abstrakter Klasse** GameCharacter
  - **Kein Wissen** über Implementierung der Subklassen nötig
  - Einfach **erweiterbar** ohne Änderung der Game Loop, z.B., Subklassen von Enemy, neue NPCs, etc.
- **Hinweis:** So ähnlich funktioniert's auch in Game Engines!

175

## Inhalt

### Vererbung

#### Zusammenfassung

176




## Zusammenfassung

- ▶ Vererbung
  - ▶ **extends**
  - ▶ Einfachvererbung, **keine** Mehrfachvererbung
  - ▶ Nicht-**private**-Methoden werden vererbt
  - ▶ Können **überschrieben** werden
  - ▶ Signatur (Parameterliste) muss **übereinstimmen**
  - ▶ Sichtbarkeit kann **größer** werden
  - ▶ Rückgabewert kann **spezieller** werden
  - ▶ Mit **super** Implementierung der Basisklasse aufrufen
  - ▶ **@Override** **nicht vergessen!**
  - ▶ **final** verhindert Überschreiben/Ableitung
- ▶ **Dynamische Bindung**: **Objekttyp** definiert aufgerufene Methode zur **Laufzeit**
- ▶ **Statische Bindung**: Aufgerufene Methode steht zur **Übersetzung** fest (**final**, **private**, **static**)




177

## Zusammenfassung

- ▶ Überschreiben der Methoden von  **Object**
  - ▶ **Aufpassen** bei `equals()` und `hashCode`
  - ▶ **super.equals()** und **super.hashCode()** aufrufen!
  - ▶ `toString()` sollte **überschrieben** werden
- ▶ Abstrakte Klassen/Methoden
  - ▶ Definieren **gemeinsame Schnittstelle** für Kategorie von Klassen
  - ▶ **erzwingen** **Überschreiben** in Ableitungen

178

## Zusammenfassung

Typ	Ableitung?	new?	Beispiel
nicht <b>final</b> / <b>abstract</b>	Ja	Ja	 game/Item.java
<b>final</b>	Nein	Ja	 game/BuffPotion.java
<b>abstract</b>	Ja	Nein	 game/GameCharacter.java

179

## Inhalt

### Interfaces

- Grundversion von interfaces
- Einschub: Das Cloneable-Interface
- Mehrere interfaces implementieren
- Erweitern von Interfaces
- Statische Elemente in interfaces
- Default-Methoden
- Bausteine über Default-Methoden
- Zusammenfassung

180

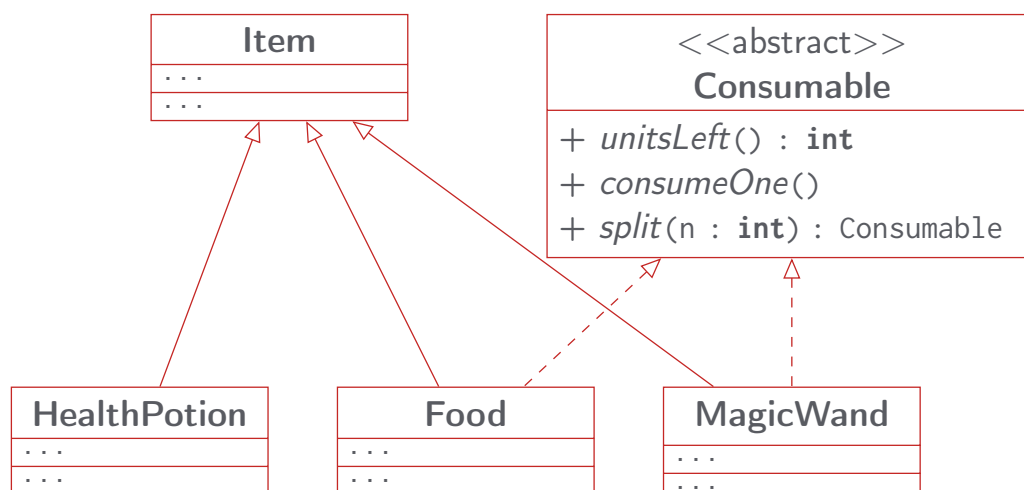
## Interfaces

### Grundversion von interfaces

- interfaces deklarieren
- Implementieren von interfaces
- interfaces im Typsystem
- Dynamische Bindung bei interfaces
- Kleine Zusammenfassung

## Warum interfaces?

- Erweiterung unseres Rollenspiel-Beispiels
  - Manche Items sind „**verbrauchbar**“: Consumable
  - **Beispiel**: Zauberstab („Ladungen“), Nahrung („Bissen“)
- Einfach noch von Consumable ableiten...
- Problem in Java: Nur **Einfachvererbung**!



## Interfaces

### Grundversion von interfaces

- interfaces deklarieren
- Implementieren von interfaces
- interfaces im Typsystem
- Dynamische Bindung bei interfaces
- Kleine Zusammenfassung

## interfaces — Grundversion

- ▶ Wie kann **Mehrfachvererbung** in Java abgebildet werden?
- ▶ Antwort: **interfaces**

```
4 public interface Consumable {  
5     int unitsLeft();  
6     void consumeOne();  
7     Consumable split(int n);  
8 }
```

game/Consumable.java

- ▶ Deklaration
  - ▶ Schlüsselwort: **interface**
  - ▶ Methodendeklarationen:
    - ▶ Rückgabewert, Name und Parameter
    - ▶ Keine Implementierung
    - ▶ Methoden sind **public** und **abstract**

## Interfaces

### Grundversion von interfaces

interfaces deklarieren

Implementieren von interfaces

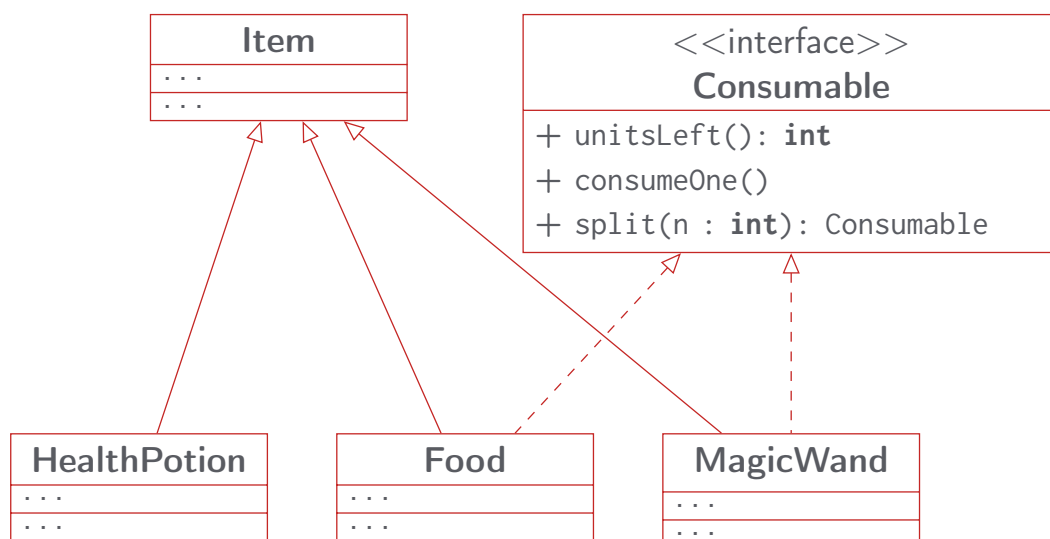
interfaces im Typsystem

Dynamische Bindung bei interfaces

Kleine Zusammenfassung

## interfaces — Grundversion

- ▶ Wie wenden wir das **interface** Consumable auf das Beispiel an?
- ▶ Food und MagicWand **implementieren** das Interface Consumable



## interfaces — Grundversion

- ▶ Was heißt „Food **implementiert** Consumable“?
- ▶ **Schlüsselwort**: **implements**

4 `public class Food extends Item implements Consumable {`

`game/Food.java`

- ▶ **Compiler-Fehler**: „Food is not abstract and does not override abstract method consumeOne() from Consumable“
- ▶ **Erkenntnisse**
  - ▶ Interface-Methoden verhalten sich wie **abstrakte Methoden**
  - ▶ Wäre Food **abstract** würde der Fehler **verschwinden**

```
public abstract class Food // kein Compiler-Fehler
    extends Item
    implements Consumable
```

187

## interfaces — Grundversion

- ▶ Wir wollen consumeOne implementieren
  - ▶ Neues Attribut bites („Bissen“)

8 `private int bites; // units left to consume`

`game/Food.java`

- ▶ Implementierung

```
@Override
void consumeOne(){
    if (bites > 0)
        bites--;
}
```

- ▶ @Override — wir **überschreiben** ja eine abstrakte Methode
- ▶ **Compiler-Fehler**: „Cannot reduce the visibility“
  - ▶ **Hier**: Paket-sichtbar
  - ▶ **In interface**: **public** (implizit)
  - ▶ Damit **immer public**

188

## interfaces — Grundversion

### ► Korrekte Version von consumeOne

```
17 @Override
18 public void consumeOne(){
19     if (bites > 0)
20         bites--;
21 }
```

game/Food.java

### ► Implementierung von unitsLeft

```
25 @Override
26 public int unitsLeft(){
27     return bites;
28 }
```

game/Food.java

189

## interfaces — Grundversion

- Welche **Regeln** gelten beim Implementieren?
- Die **gleichen** wie beim **Überschreiben** geerbter Methoden
  - **Sichtbarkeit** darf nicht kleiner werden (s. oben)
  - **Kovarianz**: Rückgabetyt darf **spezieller** sein

```
32 @Override
33 public Food split(int n){
34     if (n <= bites){
35         bites -= n;
36         return new Food(getName(), getValue(), n);
37     }else
38         throw new IllegalArgumentException("too much");
39 }
```

game/Food.java

- **Typen der Parameterliste** müssen übereinstimmen

```
@Override public Food split() // FEHLER
@Override public Food split(double n) // FEHLER
@Override public Food split(int n, int m) // FEHLER
```

190

## interfaces — Regeln

- ▶ Abstrakte Methoden und Klassen
  - ▶ Zur Erinnerung: Klassen mit einer nicht-implementierten abstrakten Methode müssen **abstract** sein
  - ▶ Interface-Methoden sind **abstract**
- ▶ Also: Implementiert eine Klasse eine **interface**-Methode nicht... **muss** die Klasse **abstract** sein
- ▶ Beispiel

```
public class InfiniteBeer extends Item
    implements Consumable{ // FEHLER
    @Override public void consumeOne(){ }
    @Override public int unitsLeft(){ return 1; }
}
```

- ▶ „InfiniteBeer is not abstract and does not implement Consumable.split(int)“
- ▶ Entweder **abstract** oder **split** implementieren
- ▶ Siehe `game/InfiniteBeer.java`

191

## MagicWand

- ▶ MagicWand: Zauberstäbe haben **Ladungen** die verbraucht werden können
  - ▶ Deklaration

```
4 public class MagicWand
5     extends Item implements Consumable {
```

`game/MagicWand.java`

- ▶ **Ladungen** als Attribut („charges“)

```
9     private int charges;
```

`game/MagicWand.java`

- ▶ **unitsLeft**

```
18 @Override
19 public int unitsLeft() {
20     return charges;
21 }
```

`game/MagicWand.java`

192



## MagicWand

- ▶ MagicWand
  - ▶ consumeOne

```
25 @Override
26 public void consumeOne() {
27     if (charges > 0)
28         charges--;
29 }
```

game/MagicWand.java

- ▶ split ? Zauberstäbe kann man nicht teilen...

```
33 @Override
34 public Consumable split(int n) {
35     throw new UnsupportedOperationException("Cannot split wand");
36 }
```

game/MagicWand.java

Informiert Aufrufer: Diese Operation wird nicht unterstützt

193

## Inhalt

### Interfaces

#### Grundversion von interfaces

interfaces deklarieren

Implementieren von interfaces

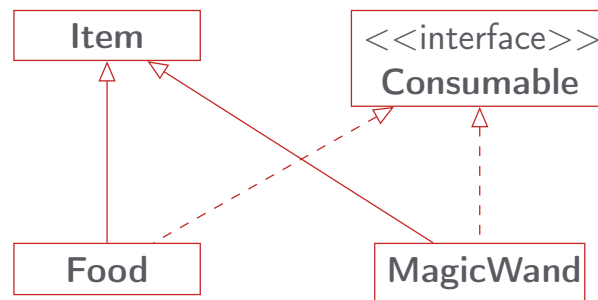
interfaces im Typsystem

Dynamische Bindung bei interfaces

Kleine Zusammenfassung

194

## interfaces im Typsystem



- ▶ **Zur Erinnerung:** **extends** definiert „ist ein“-Beziehung
  - ▶ Food **ist ein** Item
  - ▶ MagicWand **ist ein** Item
- ▶ Das gleiche gilt für **implements**
  - ▶ Food **ist ein** Consumable
  - ▶ MagicWand **ist ein** Consumable

195

## interfaces im Typsystem

- ▶ Es gelten die **gleichen Regeln** wie bei **Klassen** und **Ableitung**

```
public class Food extends Item implements Consumable
```

- ▶ Consumable definiert **Referenztyp**

```
Consumable consumable = null;
```

- ▶ Food ist **spezieller** als Consumable
- ▶ Consumable ist **allgemeiner** als Food
- ▶ **Widening-Cast** (implizit)

```
Consumable consumable = new Food(...);
```

- ▶ **Narrowing-Cast** (explizit; mit allen Konsequenzen)

```
Food food = (Food) consumable;
```

- ▶ **instanceof**

```
food instanceof Consumable // == true
```

- ▶ **Unterschied** zu Klassen: **keine direkte** Instanziierung möglich

```
Consumable c = new Consumable(); // FEHLER
```

196

## interfaces im Typsystem — Beispiel


```
13 public static void printItemInfo(Item mysteryItem) {
14     out.printf("Item: name=%s, value=%d%n",
15         mysteryItem.getName(), mysteryItem.getValue());
17     if (mysteryItem instanceof Consumable){
18         Consumable consumable = (Consumable) mysteryItem;
19         out.printf("Consumable: unitsLeft=%d%n",
20             consumable.unitsLeft());
21     } else
22         out.printf("Item is not consumable%n");
23     System.out.println();
24 }
```

BasicInterfaceExamples.java

197

## interfaces im Typsystem — Beispiel

### ► Aufruf von printItemInfo

```
31  runInterfaceTypeExample
32 var healthPotion =
33     new HealthPotion("Health Potion", 10, 100);
34 var squirrelBits =
35     new Food("Squirrel on a Stick", 2, 5);
37 printItemInfo(healthPotion);
38 printItemInfo(squirrelBits);
```

BasicInterfaceExamples.java

### ► Ausgabe

```
Item: name=Health Potion, value=10
Item is not consumable
Item: name=Squirrel on a Stick, value=2
Consumable: unitsLeft=5
```

198

## interfaces im Typsystem — Beispiel

- ▶ Beispiel für **Widening** und (gültige) **Narrowing Casts**

```
44 Food squirrelBits =  
45     new Food("Squirrel on a Stick", 2, 5);  
47 Item item = squirrelBits; // widening  
48 Consumable consumable = squirrelBits; // widening  
50 Food food = (Food) consumable; // narrowing
```

BasicInterfaceExamples.java

- ▶ Beispiel für **ungültigen Widening Cast**

```
HealthPotion potion =  
    new HealthPotion("Health Potion", 10, 100);  
Consumable consumablePotion =  
    (Consumable) potion; // FEHLER
```

HealthPotion implementiert Consumable **nicht**

199

## Inhalt

### Interfaces

#### Grundversion von interfaces

interfaces deklarieren

Implementieren von interfaces

interfaces im Typsystem

Dynamische Bindung bei interfaces

Kleine Zusammenfassung

200

## Dynamische Bindung bei interfaces

- ▶ Zur Erinnerung
  - ▶ Aufruf einer Methode, die **nicht** **static**, **final** oder **private** ist, ist **dynamisch**
  - ▶ Objekttyp bestimmt aufgerufene Methode
- ▶ Methoden aus **interfaces** sind **public** und **abstract** (**bisher**)
- ▶ Damit:
  - ▶ Methodenaufrufe auf **interface**-Methoden sind **dynamisch**
  - ▶ Objekttyp bestimmt aufgerufene Methode
- ▶ Beispiel

```
Consumable squirrelBits = new Food(...);  
Consumable fireWand = new MagicWand(...);
```

squirrelBits : Consumable

Food : ...


fireWand : Consumable

MagicWand : ...

201

## Dynamische Bindung: Beispiel

- ▶ Summiert die Einheiten der Consumables auf

```
55  runSumUnits  
56 public static int sumUnits(Consumable... consumables) {  
57     int sum = 0;  
59     for (Consumable consumable : consumables)  
60         sum += consumable.unitsLeft();  
62     return sum;  
63 }
```


 BasicInterfaceExamples.java

- ▶ Aufruf `consumable.unitsLeft()` ist **dynamisch gebunden**
- ▶ ...hängt von **Objekttyp** ab

202

# Dynamische Bindung: Beispiel

## ► Aufruf

```
68  runSumUnitsExample  
69 Food squirrelBits =  
70     new Food("Squirrel on a Stick", 2, 5);  
71 MagicWand fireWand =  
72     new MagicWand("Wand of Fire", 500, 100);  
74 var sum = sumUnits(squirrelBits, fireWand);  
76 out.printf("Sum: %d\n", sum);
```

 BasicInterfaceExamples.java

Sum: 105

- squirrelBits.unitsLeft() liefert Wert von squirrelBits.bites
- fireWand.unitsLeft() liefert Wert von fireWand.charges

203

## Inhalt

### Interfaces

#### Grundversion von interfaces

- interfaces deklarieren
- Implementieren von interfaces
- interfaces im Typsystem
- Dynamische Bindung bei interfaces
- Kleine Zusammenfassung

204

## Kleine Zusammenfassung

- ▶ Deklaration über **interfaces**. . .
  - ▶ Sammlung **abstrakter Methoden**
  - ▶ Klassen implementieren **interfaces** wie **geerbte abstrakte Methoden**
  - ▶ Ermöglicht eine Art von **Mehrfachvererbung**
- ▶ Implementierung über C **implements** I
  - ▶ Definiert C „ist ein“ I
  - ▶ Typsystem
    - ▶ C ist **spezieller** als I
    - ▶ I ist **allgemeiner** als C
  - ▶ Gleiche Regeln wie bei **Vererbung** von **abstrakten Methoden**
- ▶ Implementierte **interface**-Methode sind **dynamisch gebunden**

<<interface>> <b>Consumable</b>
+ unitsLeft(): <b>int</b>
+ consumeOne()
+ split(n : <b>int</b> ): Consumable

205

## Inhalt

### Interfaces

Einschub: Das Cloneable-Interface

206

## Object.clone()

### ► Zur Erinnerung: Wie kopieren wir Objekte?

#### ► Problem bei Kopier-Konstruktor

```
85 public static NonPlayerCharacter
86     cloneNPC(NonPlayerCharacter npc) {
87     return new NonPlayerCharacter(npc);
88 }
```

📄 ObjectOverrideExamples.java

- Funktioniert nur wenn Objekttyp von npc NonPlayerCharacter ist
- Konstruktoraufrufe können nicht dynamisch (an npc) gebunden werden
- Methodenaufrufe sind dynamisch gebunden!
- [Object](#) definiert [Object.clone\(\)](#)

207

## Object.clone()

### ► Signatur von [Object.clone\(\)](#)

```
protected Object clone()
    throws CloneNotSupportedException
```

### ► Überschreiben in Player

```
171 @Override public Player clone()
172     throws CloneNotSupportedException{
173     return (Player) super.clone();
174 }
```

📄 game/Player.java

### ► Aufruf

```
13 🐞 runCallCloneExample
14 Player player = new Player();
15 var playerClone = player.clone();
```

📄 CloneableExamples.java

Exception: [CloneNotSupportedException](#)

208



## Object.clone()

### ► Funktionsweise von `Object.clone()`

1. Prüfen ob Objekttyp von **this** `Cloneable`-Interface implementiert

```
this instanceof Cloneable
```

2. Nein: `CloneNotSupportedException` wird geworfen
3. Ja: Objekt wird geklont
  - Neues Objekt vom gleichen Objekttyp wird erstellt
  - Attribute werden über Wertzuweisung kopiert (flache Kopie)

### ► Das `Cloneable`-Interface

```
public interface Cloneable {}
```

- Keine Methoden! ???
- „Marker-Interface“: Markiert die Klasse als „klonbar“

209

## Object.clone() und Cloneable im Zusammenspiel

- Beliebt in Rollenspielen: „slime“ (Schleimkugel)
- Kann sich vervielfältigen (clone)
- Deklaration

```
4 public class SlimeBlob
5     extends Enemy implements Cloneable{
```

game/SlimeBlob.java

### ► Attribute

```
13 private int size;
14 private SlimeColor color;
```

game/SlimeBlob.java

### ► enum SlimeColor

```
9 public static enum SlimeColor{ RED, GREEN, BLUE };
```

game/SlimeBlob.java



210

## Object.clone() und Cloneable im Zusammenspiel

### ► SlimeBlob.clone()

```
33 @Override public SlimeBlob clone() {
34     try {
35         return (SlimeBlob) super.clone();
36     } catch (CloneNotSupportedException e){
37         throw new AssertionError("...");
38     }
39 }
```

game/SlimeBlob.java



### ► Hinweise

- Methode ist **public** (☞ `Object.clone` ist **protected**)
- Rückgabewert ist `SlimeBlob` (**Kovarianz**)
- **try** da ☞ `Object.clone` ☞ `CloneNotSupportedException` werfen könnte
- **catch** sollte nie auftreten, da `SlimeBlob` **implements** `Cloneable`

211

## SlimeBlobs klonen

### 21 runSlimeBlobCloneExample

```
22 var slime = new SlimeBlob("Large Green Slime",
23     1, 2, 100, 20, 50, SlimeBlob.SlimeColor.GREEN);
25 var slimeClone = slime.clone();
27 out.printf("slime == slimeClone: %b\n", slime == slimeClone);
28 out.println(slime);
29 out.println(slimeClone);
```

CloneableExamples.java

```
slime == slimeClone: false
SlimeBlob: name="Large Green Slime", health=100,
  x=1, y=2, attackPower=20, size=50, color=GREEN
SlimeBlob: name="Large Green Slime", health=100,
  x=1, y=2, attackPower=20, size=50, color=GREEN
```

212

## Hinweise zu Cloneable und Object.clone

- ▶ `Object.clone()` erstellt eine **flache** Kopie
- ▶ Für **tiefe** Kopien braucht es mehr Arbeit
- ▶ **Beispiel**: neues Attribute parent in SlimeBlob

```
private SlimeBlob parent;
```

- ▶ Klon referenziert **denselben** parent
- ▶ **Tiefe Kopie**

```
try{  
    var clone = (SlimeBlob) super.clone();  
    clone.parent = (SlimeBlob) parent.clone();  
    return clone;  
} catch (...)
```

- ▶ **Voraussetzungen**: Referenzierte Objekttypen müssen
  - ▶ `Cloneable` sein
  - ▶ müssen **tiefe Kopie** implementieren

213

## Inhalt

### Interfaces

Mehrere interfaces implementieren

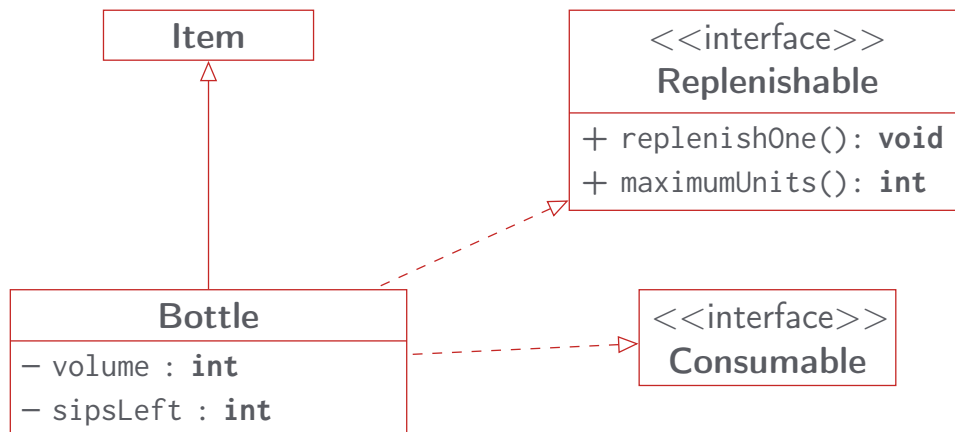
Kompatible Interfaces

Inkompatible Interfaces

214

## Mehrere interfaces implementieren

- ▶ Manche Items sollen auch wieder **aufgefüllt** werden können
- ▶ **interface** Replenishable („wiederbefüllbar“)
- ▶ **Neues Item** Bottle für Wasser, Zaubertränke, etc.
  - ▶ Consumable — Inhalt entnehmbar
  - ▶ Replenishable — wiederbefüllbar
- ▶ Bottle kann **beide** interfaces implementieren



215

## Mehrere interfaces implementieren

- ▶ Interface Replenishable

```
4 public interface Replenishable{
5     void replenishOne();
6     int maximumUnits();
7 }
```

game/Replenishable.java

- ▶ Deklaration

```
4 public class Bottle extends Item
5     implements Consumable, Replenishable{
```

game/Bottle.java

- ▶ Attribute

```
9 private int sipsLeft;
10 private int volume;
```

game/Bottle.java

216

## Mehrere interfaces implementieren

- ▶ Interface Consumable
  - ▶ consumeOne und unitsLeft wie bei MagicWand und Food
  - ▶ split wird **nicht unterstützt** (↗ `UnsupportedOperationException`)
- ▶ Interface Replenishable
  - ▶ replenishOne

```
42 @Override
43 public void replenishOne() {
44     if (sipsLeft < volume)
45         sipsLeft++;
46 }
```

game/Bottle.java

- ▶ maximumUnits

```
51 @Override
52 public int maximumUnits() {
53     return volume;
54 }
```

game/Bottle.java

217

## Beispiel

- ▶ printInfo


```
15 public static void printInfo(Item item) {
16     out.printf("Name: %s\n", item.getName());
17
18     if (item instanceof Consumable){
19         var consumable = (Consumable) item;
20         out.printf("units left: %d\n", consumable.unitsLeft());
21     } else out.println("Not Consumable");
22
23     if (item instanceof Replenishable){
24         var replenishable = (Replenishable) item;
25         out.printf("maximum units: %d\n", replenishable.maximumUnits());
26     } else out.println("Not Replenishable");
27
28     out.println();
29 }
```

MultipleInterfacesExamples.java

218

## Beispiel

### ► Aufruf

```
34  runMultipleInterfacesExample  
35 var fairyBottle = new Bottle("Fairy Bottle", 10, 10, 2);  
36 var squirrelBits = new Food("Squirrel on a Stick", 2, 5);  
38 printInfo(fairyBottle);  
39 printInfo(squirrelBits);
```

 MultipleInterfacesExamples.java

### ► Ausgabe

```
Name: Fairy Bottle  
units left: 2  
maximum units: 10  
Name: Squirrel on a Stick  
units left: 5  
Not Replenishable
```

219

## Inhalt

### Interfaces

Mehrere interfaces implementieren

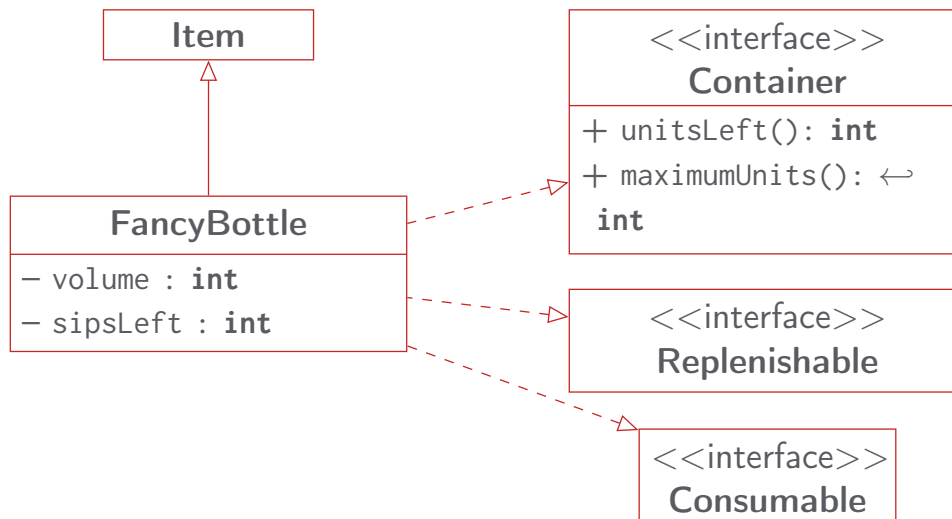
Kompatible Interfaces

Inkompatible Interfaces

220

## Konflikte beim Implementieren mehrerer Interfaces

- ▶ Neues **interface** Container liefert Informationen über Inhalt
  - ▶ **int** unitsLeft() — liefert Inhalt
  - ▶ **int** maximumUnits() — liefert max. Inhalt
- ▶ FancyBottle implementiert **alle drei Interfaces**



221

## Konflikt (?)

- ▶ Interface Container

```
4 public interface Container {
5     int unitsLeft();
6     int maximumUnits();
7 }
```

game/Container.java

- ▶ Klasse FancyBottle

```
4 public class FancyBottle extends Item
5     implements Consumable, Replenishable, Container{
```

game/FancyBottle.java

- ▶ „Konflikt“

- ▶ Consumable definiert ebenfalls **int** unitsLeft()
- ▶ Replenishable definiert ebenfalls **int** maximumUnits()
- ▶ Signatur jeweils **identisch**: gleicher Name, gleiche (leere) Parameterliste

222

## Kein Konflikt!

- ▶ Es gibt **keinen** Konflikt!
  - ▶ Consumable sagt: „Implementiere **int** unitsLeft()“!
  - ▶ Replenishable sagt: „Implementiere **int** maximumVolume()“!
  - ▶ Container sagt: „Implementiere **beide** genau so“!
- ▶ Signaturen **widersprechen** sich **nicht**
- ▶ Implementierung „befriedigt“ **alle** Anforderungen

```
35 @Override
36 public int unitsLeft() {
37     return sipsLeft;
38 }
```

game/FancyBottle.java

```
51 @Override
52 public int maximumUnits() {
53     return volume;
54 }
```

game/FancyBottle.java

223

## Beispiel

- ▶ printConsumableInfo

```
44 public static void
45     printConsumableInfo(Consumable consumable) {
46     out.printf("Consumable.unitsLeft(): %d\n",
47         consumable.unitsLeft());
48 }
```

MultipleInterfacesExamples.java

- ▶ printReplenishableInfo

```
52 public static void
53     printReplenishableInfo(Replenishable replenishable) {
54     out.printf("Replenishable.maximumUnits(): %d\n",
55         replenishable.maximumUnits());
56 }
```

MultipleInterfacesExamples.java

224




## Beispiel

### ► printContainerInfo

```
60 public static void
61     printContainerInfo(Container container) {
62         out.printf("Container percent full: %f%n",
63             (100f * container.unitsLeft()) / container.maximumUnits());
64     }
```

MultipleInterfacesExamples.java

### ► Aufruf

```
69  runMultipleInterfacesExample2
70 var fancyBottle =
71     new FancyBottle("Fancy Bottle", 10, 10, 2);
73 printConsumableInfo(fancyBottle);
74 printReplenishableInfo(fancyBottle);
75 printContainerInfo(fancyBottle);
```

MultipleInterfacesExamples.java

225

## Beispiel

### ► Ausgabe

```
Consumable.unitsLeft(): 2
Replenishable.maximumUnits(): 10
Container percent full: 20,000000
```

### ► Hinweise

- Das ist **Polymorphie** in Reinform!
- Obwohl **Bottle** `unitsLeft` und `maximumUnits` implementiert, ist **Bottle** **nicht kompatibel** mit **Container**

```
Container c = new Bottle(); // FEHLER
```

„Cannot convert from Bottle to Container“

226

## Interfaces

### Mehrere interfaces implementieren

Kompatible Interfaces

Inkompatible Interfaces

## Ein echter Konflikt

- ▶ Rückgabewerte von Methoden gleicher Signatur und unterschiedlicher Interfaces müssen kompatibel sein
- ▶ Variante von Container

```
public interface Container {  
    double unitsLeft();  
    double maximumUnits();  
}
```

- ▶ FancyBottle.unitsLeft(): Was ist der Rückgabewert?
  - ▶ **int?**

```
@Override public int unitsLeft() // FEHLER
```

„The return type is incompatible with Container.unitsLeft()“

- ▶ **double?**

```
@Override public double unitsLeft() // FEHLER
```

„The return type is incompatible with Consumable.unitsLeft()“

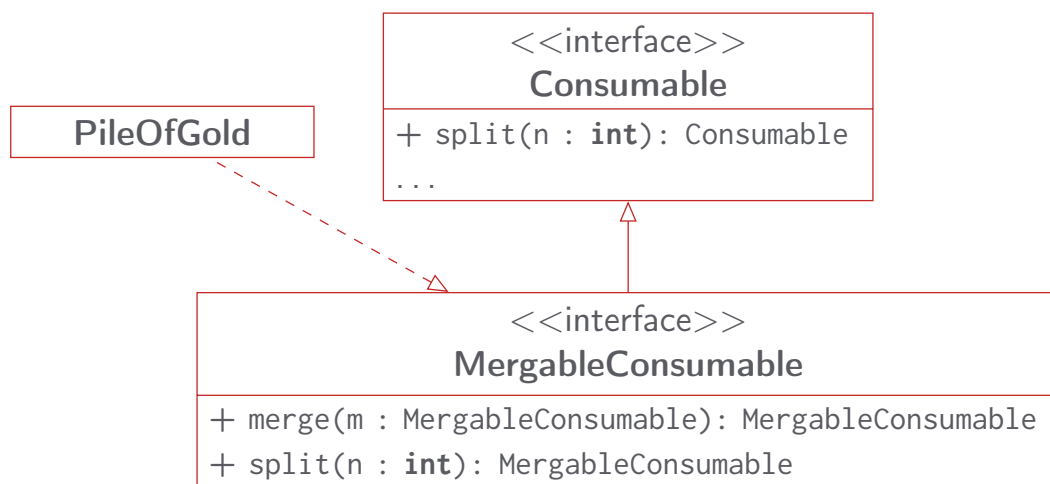
- ▶ Entsprechend bei maximumVolume
- ▶ In diesem Fall sind die Schnittstellen nicht kompatibel!

## Interfaces

### Erweitern von Interfaces

## Erweitern von Interfaces

- ▶ Interfaces können **erweitert** werden zu **Subinterfaces**
- ▶ **Neues Interface** MergableConsumable erweitert Consumable um merge (**zusammenführen**)
- ▶ PileOfGold („Goldhaufen“) implementiert MergableConsumable



## MergableConsumable

### ► Interface MergableConsumable

```
4 public interface MergableConsumable
5     extends Consumable {
6     MergableConsumable merge(MergableConsumable other);
7     @Override MergableConsumable split(int n);
8 }
```

game/MergableConsumable.java

- **Syntax:** Wie bei **Klassenvererbung**
- **Neue** Methode merge
- **Kovarianz:** split ist **überschrieben** und hat **spezielleren** Rückgabotyp
- **Allgemein:** Gleiche Regeln wie bei **Klassenvererbung**
  - „ist ein“-Beziehung: MergableConsumable **ist ein** Consumable
  - Regeln beim **Überschreiben** von Methoden
    - Typ darf **spezieller** werden
    - gleiche **Signatur**
    - @Override **nicht vergessen!**

231

## PileOfGold

### ► Deklaration von PileOfGold

```
4 public class PileOfGold extends Item
5     implements MergableConsumable{
```

game/PileOfGold.java

### ► Anzahl Münzen coins

```
9 private int coins;
```

game/PileOfGold.java

### ► split

```
40 @Override public PileOfGold split(int n) {
```

game/PileOfGold.java

- Implementierung wie in Food
- **Kovarianz:** Rückgabotyp PileOfGold ist spezieller MergableConsumable

232

## PileOfGold

- merge vereinigt mit anderem PileOfGold und gibt **neuen** PileOfGold zurück


```
29 @Override
30 public PileOfGold merge(MergableConsumable other) {
31     if (other instanceof PileOfGold){
32         var otherPile = (PileOfGold) other;
33         return new PileOfGold(coins + otherPile.coins);
34     } else
35         throw new IllegalArgumentException("other must be PileOfGold");
36 }
```

game/PileOfGold.java

- Nur erlaubt wenn other auch PileOfGold ist
- **Kovarianz** wie bei split

233

## Beispiel

```
9   runSubinterfaceExample
10 PileOfGold smallPile = new PileOfGold(10);
11 PileOfGold bigPile = new PileOfGold(100);
13 PileOfGold biggerPile = smallPile.merge(bigPile);
15 System.out.printf("Coins in bigger pile: %d\n",
16     biggerPile.unitsLeft());
18 PileOfGold mediumPile = biggerPile.split(60);
20 System.out.printf("Coins in medium pile: %d\n",
21     mediumPile.unitsLeft());
```

SubinterfaceExamples.java

```
Coins in bigger pile: 110
Coins in medium pile: 60
```

- **Schön**: Durch **Kovarianz** kein Cast bei merge/split nötig

234

## Interfaces

### Statische Elemente in interfaces

## Statische Elemente in interfaces

- ▶ Interfaces können **static** Elemente enthalten
  - ▶ **final** Konstanten
  - ▶ Statische Methoden
- ▶ **Beispiel:** Version von `game/Container.java` mit statischen Elementen

```
4 public interface Container2 {  
6     double HUNDRED_PERCENT = 100.0;  
8     static double percentFull(Container2 container){  
9         return (HUNDRED_PERCENT * container.unitsLeft())  
10        / container.maximumUnits();  
11    }  
13    int unitsLeft();  
14    int maximumUnits();  
15 }
```

`game/Container2.java`

## Statische Elemente in interfaces

### ► Konstante

```
double HUNDRED_PERCENT = 100.0;
```

- **public static final** sind **implizit**
- **Zugriff von außen** über Interface-Namen

```
Container2.HUNDRED_PERCENT
```

### ► Methode

```
static double percentFull(Container2 container){  
    return (HUNDRED_PERCENT * container.unitsLeft())  
        / container.maximumUnits();  
}
```

- **public** implizit
- **Zugriff von außen** auch über Interface-Namen

```
Container2.percentFull(c);
```

237

## FancierBottle

### ► FancierBottle

```
4 public class FancierBottle  
5     extends FancyBottle implements Container2 {  
7     public FancierBottle(String name, int value,  
8         int volume, int sipsLeft){  
9         super(name, value, volume, sipsLeft);  
10    }  
12    public double percentFull(){  
13        return Container2.percentFull(this);  
14    }  
15 }
```

game/FancierBottle.java

- Erbt von game/FancyBottle.java
- Methode percentFull nutzt Hilfsmethode Container2.percentFull

238

## Hinweise

- ▶ Nützlich für **Hilfsmethoden**
- ▶ Oder üblicher: **final**-Klassen wie [Math](#)
- ▶ **Achtung**
  - ▶ Statische Interface Methoden werden **nicht** vererbt

```
public interface Container3 extends Container2{ }
```

```
Container3.percentFull(c); // FEHLER
```

- ▶ Konstanten werden **vererbt**

```
Container3.HUNDRED_PERCENT; // == 100.0 OK
```

- ▶ Können aber **verschattet** werden

```
public interface Container3 extends Container2{  
    double HUNDRED_PERCENT = 200.0; // O_o  
}
```

```
Container2.HUNDRED_PERCENT // == 100.0  
Container3.HUNDRED_PERCENT // == 200.0
```

239

## Inhalt

### Interfaces

#### Default-Methoden

- Nachträgliches Ändern von interfaces
- Default-Methoden
- Überschreiben von Default-Methoden
- Konflikte
- private Default-Methoden und Konstanten

240



## Interfaces

### Default-Methoden

- Nachträgliches Ändern von interfaces
- Default-Methoden
- Überschreiben von Default-Methoden
- Konflikte
- private Default-Methoden und Konstanten

## Nachträgliches Ändern von interfaces

- **Zur Erinnerung:** `Consumable.consumeOne()` konsumiert eine Einheit
- **Aber:** Was ist wenn man **mehr** Einheiten auf einmal konsumieren will?

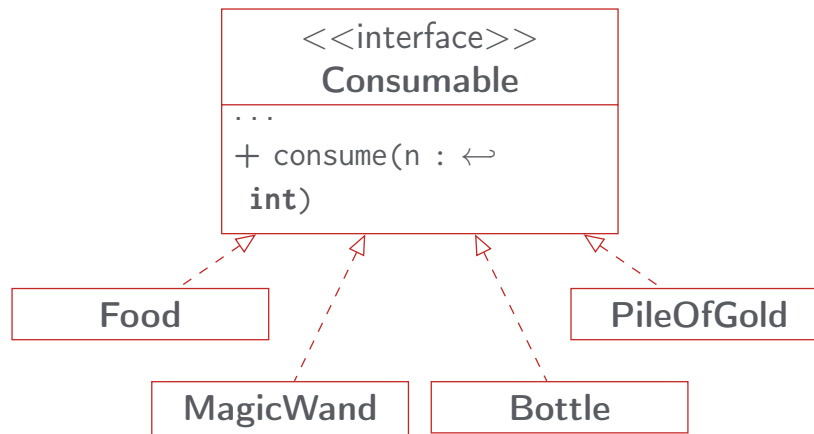
```
var squirrelBits = new Food(...);  
for (int i = 0; i < 1000; i++)  
    squirrelBits.consumeOne();
```

- **Unschön und langsam**
- **Idee:** Wir führen eine **neue Methode** `consume(int n)` in `Consumable` ein!

```
public interface Consumable {  
    int unitsLeft();  
    void consumeOne();  
    Consumable split(int n);  
    void consume(int n); // NEU  
}
```

## Nachträgliches Ändern von interfaces

- Keine gute Idee...



- **Problem:** Keine der Klassen **übersetzt** mehr
- **Grund:** Keine implementiert `consume`
- Stellen Sie sich vor, das wäre eine API, die von ganz **vielen Softwareprojekten** verwendet wird...

243

## Lösungsversuch

- Vorheriges Problem tritt oft auf
  - Refactoring
  - Sogar im **JDK**
- **Lösungsansatz:** Interface das Consumable um neue Methode erweitert

```
public interface BetterConsumable extends Consumable{
    void consume(int n); // NEU
}
```

- **Vorteil:** Obige Klassen übersetzen wieder
- **Nachteil:** Neue Schnittstelle kann **noch** nicht genutzt werden (**Refactoring** trotzdem notwendig)
- Nur eine **Notlösung**
- **Grundsätzliche Lösung** notwendig: Default-Methoden

244

## Interfaces

### Default-Methoden

- Nachträgliches Ändern von interfaces
- Default-Methoden
- Überschreiben von Default-Methoden
- Konflikte
- private Default-Methoden und Konstanten

## Default-Methoden

### ► Default-Methoden

```
public interface I {  
    default void m(){ ... }  
}
```

- Modifizierer **default**
- Nicht **abstract**: Beinhaltet Implementierung
- Erlauben **Erweiterung** von Interfaces mit einer **Default-Implementierung**
- Vorteile
  - Alter Code **übersetzt** immer noch
  - Neue Schnittstelle kann **sofort** verwendet werden
- Nachteil
  - Implementierung von Default-Methode muss evtl. **viele Kompromisse** eingehen
  - Letztendlich immer noch **Refactoring** notwendig

## Default-Methoden: Beispiel

- Idee: Wir erweitern Consumable um Default-Methode consume(**int**)

```
4 public interface Consumable {
5     int unitsLeft();
6     void consumeOne();
7     Consumable split(int n);
8
9     default void consume(int n){
10         for (int i = 0; i < n; i++)
11             consumeOne();
12     }
13 }
```


gamerefactored/Consumable.java

- Hinweise
  - **Kontext** der Default-Methode ist das Interface
  - Insbesondere ist **this** ist **Referenz auf Interface**

247

## Default-Methoden: Beispiel

- Verwendung

```
11  runDefaultMethodExample
12 Food squirrelBits =
13     new Food("Squirrel on a Stick", 10, 10000);
14
15 out.printf("Bites: %d\n", squirrelBits.unitsLeft());
16 squirrelBits.consume(1000);
17 out.printf("Bites: %d\n", squirrelBits.unitsLeft());
```

DefaultMethodExamples.java

```
Bites: 10000
Bites: 9000
```

- Vorteil:
  - Implementierung von Food wurde **nicht** verändert
  - Neues Feature kann **sofort** verwendet werden
- Nachteil: Implementierung **ungünstig** (langsam)

248

## Interfaces

### Default-Methoden

Nachträgliches Ändern von interfaces  
Default-Methoden  
Überschreiben von Default-Methoden  
Konflikte  
private Default-Methoden und Konstanten

## Überschreiben von Default-Methoden

- Default-Methoden können überschrieben werden
  - Es gelten die **bekannten Regeln** beim **Überschreiben**
  - In **implementierender Klassen**, z.B. MagicWand

```
33 @Override
34 public void consume(int n){
35     if (charges >= n)
36         charges -= n;
37 }
```

gamerefactored/MagicWand.java

- In **Subinterface**, z.B., MergableConsumable

```
10 @Override default void consume(int n){
11     int i = 0;
12     while (i < n){
13         consumeOne();
14         i++;
15     }
16 }
```

gamerefactored/MergableConsumable.java

# Überschreiben von Default-Methoden

- **Hinweis:** Ableitende Interfaces können Default-Methoden sogar wieder **abstract** machen

```
4 public interface AbstractConsumable
5     extends Consumable {
6     @Override void consume(int n);
7 }
```

gamerefactored/AbstractConsumable.java

- **Noch ein Hinweis:** Klassen-Implementierung hat **Vorrang** vor Default-Methoden

- **Annahme:** Item implementiert consume(int n)
- Food **extends** Item **implements** Consumable
- Welche Implementierung wird verwendet?

```
var squirrelBits = new Food(...);
squirrelBits.consume(10); // Item oder Consumable?
```

- Implementierung der Klasse (Item)
- **Grund:** Rückwärtskompatibilität, Default-Methoden wurden erst **später** eingeführt

251

## Inhalt

### Interfaces

#### Default-Methoden

Nachträgliches Ändern von interfaces  
Default-Methoden  
Überschreiben von Default-Methoden  
Konflikte  
private Default-Methoden und Konstanten

252

## Konflikte

- Was passiert wenn eine Klasse Default-Methoden mit **gleicher Signatur** „erbt“?

- **Neue Version** von **interface** Replenishable

```
4 public interface Replenishable{
5     void replenishOne();
6     default int maximumUnits() {
7         return Integer.MAX_VALUE;
8     }
9 }
```

gamerefactored/Replenishable.java

- **Neue Version** von **interface** Container

```
4 public interface Container {
5     int unitsLeft();
6     default int maximumUnits(){
7         return 10;
8     }
9 }
```

gamerefactored/Container.java

253

## Konflikte

- Default-Methode für **int** maximumUnits() in **beiden** Interfaces
- Was passiert wenn eine Klasse **beide Interfaces** implementiert?

```
public class FancyBottle extends Item
    implements Consumable, Replenishable, Container{
}
```

Zwei Fälle

- FancyBottle überschreibt maximumUnits **nicht**
  - **Konflikt** führt zu Compiler-Fehler
  - „Duplicate default methods are inherited“
- FancyBottle überschreibt maximumUnits

```
51 @Override
52 public int maximumUnits() {
53     return volume;
54 }
```

gamerefactored/FancyBottle.java

**Kein Konflikt:** Diese Implementierung hat **Vorrang**

254

## Konflikt

- ▶ Beim Überschreiben kann auch auf **Default-Methoden** zurückgegriffen werden
- ▶ Zugriff über

```
InterfaceName.super.defaultMethode()
```

- ▶ **Beispiel**

- ▶ Zugriff auf `Replenishable.maximumUnits()` in `FancyBottle`:

```
@Override public int maximumUnits() {  
    return Replenishable.super.maximumUnits();  
}
```

- ▶ **Oder**: Zugriff auf `Container.maximumUnits()` in `FancyBottle`:

```
@Override public int maximumUnits() {  
    return Container.super.maximumUnits();  
}
```

- ▶ In jedem Fall: Methode **muss** überschrieben werden

255

## Inhalt

### Interfaces

#### Default-Methoden

- Nachträgliches Ändern von interfaces
- Default-Methoden
- Überschreiben von Default-Methoden
- Konflikte
- private Default-Methoden und Konstanten

256



## private Methoden und Konstanten

- ▶ **interfaces** können **private** Elemente enthalten
  - ▶ **static** und **default** Methoden
  - ▶ Konstanten (**final**)
- ▶ **Eigenschaften**
  - ▶ Werden **nicht** vererbt
  - ▶ Default-Methoden sind **statisch** gebunden
- ▶ Hilfreich für **interne Hilfsmethoden**

257

## private Default-Methoden und Konstanten: Beispiel

```
public interface Container{
    private final HUNDRED_PERCENT = 100.0;
    private default float computePercent(double x, double y){
        return HUNDRED_PERCENT * x / y;
    }
    private static double computePercentLeft(Container c){
        return computePercent(
            c.unitsLeft(), c.maximumUnits());
    }
    int unitsLeft();
    maximumUnits();
    default double percentLeft(){
        return computePercentLeft(this);
    }
}
```

258

## Interfaces

### Bausteine über Default-Methoden

## Bausteine über Default-Methoden

- ▶ Default-Methoden lassen sich zum Erstellen von „**Programm-Bausteinen**“ nutzen
- ▶ **Beispiel**

- ▶ **Logging**: Ausgabe von Nachrichten in Datei/Konsole

```
out.printf("INFO: Method XYZ called");
```

- ▶ Logging-Code ist **unabhängig** zur restlichen Logik
- ▶ Wie könnte man eine Klasse um Logging schnell erweitern?
  - ▶ Möglichst **geringer Aufwand**
  - ▶ Möglichst **unabhängig** von restlicher Logik
- ▶ **Idee**:
  - ▶ Wir implementieren Logging-Funktionalität in **Default-Methoden** eines Interfaces
  - ▶ Klassen, die Logging brauchen, „**implementieren**“ das Interface

## Bausteine über Default-Methoden

### ► Interface Logged

```
7 public interface Logged{
8     default void info(String message){
9         out.printf("INFO (%s): %s\n",
10             this.getClass().getSimpleName(), message);
11     }
12
13     default void error(String message){
14         err.printf("ERROR (%s): %s\n",
15             this.getClass().getSimpleName(), message);
16     }
17 }
```

gamerefactored/Logged.java

### ► Beachte

- Das Interface hat nur **zwei Default-Methoden**
- ... es besitzt keine **abstract** Methoden

261

## Bausteine über Default-Methoden

### ► Die Klasse Bottle will Logging nutzen

```
4 public class Bottle extends Item
5     implements Consumable, Replenishable, Logged{
```

gamerefactored/Bottle.java

### ► Sie kann jetzt info und error direkt nutzen

```
20 @Override public void consumeOne() {
21     info("consumeOne called");
22     if (sipsLeft > 0) sipsLeft--;
23     else error("bottle empty");
24 }
```

gamerefactored/Bottle.java


```
42 @Override public void replenishOne() {
43     info("replenishOne called");
44     if (sipsLeft < volume) sipsLeft++;
45     else error("bottle full");
46 }
```

gamerefactored/Bottle.java

262

## Beispiel

- ▶ Aufruf auf Bottle mit einem übrigen Schluck

```
10  runTraitsExample  
11 var bottle = new Bottle("Water Bottle", 0, 10, 1);  
13 bottle.consumeOne();  
14 bottle.consumeOne();  
15 bottle.replenishOne();
```

 TraitsExamples.java

- ▶ Ausgabe

```
INFO (Bottle): consumeOne called  
INFO (Bottle): consumeOne called  
ERROR (Bottle): bottle empty  
INFO (Bottle): replenishOne called
```

- ▶ Vorteile: Logging-Funktion kann einfach **angebaut** werden

263

## Inhalt

### Interfaces

#### Zusammenfassung

264

## Zusammenfassung

### ► interfaces

```
public interface Consumable{ }
```

- **Trennung**: Schnittstelle von Code
- Definieren **Anforderungen**: Was muss implementiert werden
- **Methoden** sind **abstract**
- **interface** definiert **Referenztyp**

### ► Food **implements** Consumable

```
4 public class Food extends Item implements Consumable {
```

game/Food.java

- **Typsystem**: Definiert „ist ein“-Beziehung
- Klassen **implementieren interface** (das „Wie“), indem...
- sie die Interface-Methoden **überschreiben**

265

## Zusammenfassung

### ► Klassen können **mehrere Interface** implementieren

```
4 public class FancyBottle extends Item  
5 implements Consumable, Replenishable, Container{
```

game/FancyBottle.java

- **Kein Problem**: Gleiche Signaturen bei **kompatiblen** Rückgabewerten

### ► Interfaces können andere Interfaces **erweitern**

```
public interface MergableConsumable  
extends Consumable { }
```

- **Einführung** neuer Methoden
- **Überschreiben** geerbter Methoden (z.B. **Kovarianz**)
- Interfaces können **static** Elemente enthalten
  - **static** Methoden (Hilfsmethoden)
  - **final** Konstanten

266

# Zusammenfassung

- ▶ Interfaces können **private** Elemente enthalten
  - ▶ **static** und **default** Methoden (Hilfsmethoden)
  - ▶ **final** Konstanten
- ▶ Default-Methoden

```
default void consume(int n){ }
```

- ▶ Geben **Default-Implementierung** einer Methode vor
- ▶ Nützlich bei **Änderung** von Schnittstellen
- ▶ **Und**: Zur Entwicklung von **Bausteinen**
- ▶ **Konflikt** bei Default-Methoden gleicher Signatur, unterschiedlicher Interfaces
  - ▶ Klasse **muss** Methode überschreiben und...
  - ▶ kann über **super** Default-Methoden aufrufen

267

## Inhalt

### Geschachtelte Typen

- Statische geschachtelte Typen
- Innere Typen
- Lokale Klassen
- Anonyme Klassen
- Zusammenfassung

268

## Geschachtelte Typen

### Statische geschachtelte Typen

## Statische geschachtelte Typen

- ▶ „Top-Level Typen“ (**class**, **interface**, **enum**) können **geschachtelte** Typen beinhalten

```
public class/interface/enum TopLevel{  
    public static class/interface/enum Nested{  
        ...  
    }  
}
```

- ▶ Zugriff wie **static** Elemente

```
TopLevel.Nested nested = new TopLevel.Nested();
```

- ▶ Besonderheiten

- ▶ Top-Level Typen können nur **public** oder **Paket-sichtbar** sein
- ▶ Geschachtelte Typen können auch **private** oder **protected** sein

- ▶ Verwendung:

- ▶ Top-Level Typ definiert eigenen **Namensraum**
- ▶ **private/protected** für Typen, die nach außen **nicht sichtbar** sein sollen

## Beispiel: SlimeBlob.SlimeColor

- ▶ `enum SlimeColor` ist in **SlimeBlob** geschachtelt

```
public class SlimeBlob ... {  
    public static enum SlimeColor{ RED, GREEN, BLUE };  
}
```

- ▶ **Hinweis:** Geschachtelte `enums` sind immer `static` (`static` optional)
- ▶ Grund: **starke Zugehörigkeit** zu `SlimeBlob`
- ▶ **Verwendung**

```
SlimeBlob.SlimeColor c = SlimeBlob.SlimeColor.RED;
```

- ▶ Geht auch mit `interface`

```
public class SlimeBlob ... {  
    public static interface Slimable{  
        void slime();  
    }  
}
```

- ▶ Zugriff wie bisher

```
public class SuperSlime implements SlimeBlob.Slimable
```

271

## Beispiel: SlimeBlob.Blobling

- ▶ `SlimeBlob` soll sich in **zwei Kinder halber Größe teilen** können
- ▶ Klasse für Kinder **geschachtelt** in `SlimeBlob`

```
72 private static class Blobling extends SlimeBlob{  
73     private final SlimeBlob parent;  
75     private Blobling(SlimeBlob parent, String name,  
76         int x, int y, int health, int attackPower,  
77         int size, SlimeColor color){  
78         super(name,x,y,health,attackPower,size,color);  
79         this.parent = parent;  
80     }  
82     public SlimeBlob getParent() { return parent; }  
83 }
```

game/SlimeBlob.java

- ▶ **private** Klasse und Konstruktor (nur in `SlimeBlob` sichtbar)
- ▶ Referenz auf Eltern-`SlimeBlob`

272



## Beispiel: SlimeBlob.Blobling

### ► SlimeBlob.divide


```
43 public SlimeBlob[] divide(){
44     Blobling child1 =
45         new Blobling(this, getName()+"ling 1",
46             getX(), getY(), getHealth(),
47             getAttackPower(), size/2, color);
49     Blobling child2 =
50         new Blobling(this, getName()+"ling 2",
51             getX(), getY(), getHealth(),
52             getAttackPower(), size/2, color);
54     return new SlimeBlob[] { child1, child2 };
55 }
```

game/SlimeBlob.java

- Erstellt zwei Bloblings mit halber Größe
- Referenztyp: SlimeBlob
- Objekttyp: SlimeBlob.Blobling (**private**)

273

## Beispiel: SlimeBlob.Blobling

```
12  runBloblingsExample
13 out.println(slimeBlob);
15 SlimeBlob[] children = slimeBlob.divide();
16 out.println(children[0]);
17 out.println(children[1]);
```

NestedTypesExamples.java

```
SlimeBlob: name="Blob", health=100, x=0, y=1, attackPower=20, size=60, color=RED
Blobling: name="Blobling 1", health=100, x=0, y=1, attackPower=20, size=30, color=RED
Blobling: name="Blobling 2", health=100, x=0, y=1, attackPower=20, size=30, color=RED
```

- **Beachte:** Objekttyp der Kinder ist Blobling

274

## Hinweise

- ▶ Auch **interfaces** und **enums** können Typen schachteln

```
public interface Slimable{  
    public static enum Sliminess { NOT_VERY, DISGUSTING };  
}
```

- ▶ Geschachtelte Typen können auf **alle** Attribute des übergeordneten Typs zugreifen...
- ▶ wenn Sie eine **Referenz** darauf haben
- ▶ **Beispiel**: SlimeBlob.Blobling

```
private void printSize(){  
    out.println(parent.size);  
}
```

- ▶ Alles in SlimeBlob ist für SlimeBlob.Blobling **sichtbar**
- ▶ **Hier**: Beziehung Blobling zu SlimeBlob über explizite Referenz
- ▶ **Verbindung** „Objekt ↔ Objekt geschachtelter Typ“ existiert **oft**
- ▶ **Allgemeine Lösung**: Innere Typen

275

## Inhalt

### Geschachtelte Typen

#### Innere Typen

276

## Innere Typen

- ▶ Geschachtelte Typen können auch **nicht-static** sein

```
public class/interface/enum TopLevel {  
    public class/interface Nested { }  
}
```

- ▶ **Unterschied** zu **static**-Variante
  - ▶ Instanziierung **nur** über **Instanz** von TopLevel möglich
  - ▶ Instanz von Nested ist an **umschließende Instanz** gebunden
  - ▶ ... und hat eine (versteckte) **Referenz** darauf
- ▶ **Zugriff**

```
TopLevel t = new TopLevel();  
TopLevel.Nested n = t.new Nested(); // O_o
```

- ▶ **Instanz** von TopLevel notwendig
  - ▶ n ist an t gekoppelt
- ▶ **Kein new** über Klasse mehr möglich

```
TopLevel.Nested n = new TopLevel.Nested(); // FEHLER
```

„An enclosing instance is required“

277

## SlimeBlob.InnerBlobling

- ▶ Beispiel von vorher: SlimeBlob.Blobling
  - ▶ **static**
  - ▶ Bekam **Referenz** auf SlimeBlob parent
- ▶ Jetzt **innere Klasse** SlimeBlob.InnerBlobling

```
87 public class InnerBlobling extends SlimeBlob{  
89     public InnerBlobling(String name,  
90         int x, int y, int health, int attackPower,  
91         int size, SlimeColor color){  
92         super(name,x,y,health,attackPower,size,color);  
93     }  
95     public int getParentSize() {  
96         return SlimeBlob.this.size;  
97     }  
98 }
```

game/SlimeBlob.java

278

## SlimeBlob.innerDivide

- Entsprechende Variante von SlimeBlob.divide


```
59 public SlimeBlob[] innerDivide(){
60     InnerBlobling child1 =
61         new InnerBlobling(getName()+"ling 1", getX(), getY(),
62             getHealth(), getAttackPower(), size/2, color);
63     InnerBlobling child2 =
64         new InnerBlobling(getName()+"ling 2", getX(), getY(),
65             getHealth(), getAttackPower(), size/2, color);
67     return new SlimeBlob[] { child1, child2 };
68 }
```

game/SlimeBlob.java

- Hinweis `new InnerBlobling(...)` entspricht **eigentlich** `this.new InnerBlobling(...)`

279

## Ausführen von SlimeBlob.innerDivide

```
25  runInnerBlobsExample
26 out.println(slimeBlob);
28 SlimeBlob[] children = slimeBlob.innerDivide();
29 out.println(children[0]);
30 out.println(children[1]);
```

NestedTypesExamples.java

```
SlimeBlob: name="Blob", health=100, x=0, y=1, attackPower=20, size=60, color=RED
InnerBlobling: name="Blobling 1", health=100, x=0, y=1, attackPower=20, size=30, ↵
    color=RED
InnerBlobling: name="Blobling 2", health=100, x=0, y=1, attackPower=20, size=30, ↵
    color=RED
```

**Beachte:** Typ der Kinder ist nun InnerBlobling

280

## Blobling vs. InnerBlobling

- ▶ Eine Instanz von `SlimeBlob.InnerBlobling` ist an **umschließende Instanz** gebunden
- ▶ Umschließende Instanz kann über `KlassenName.this` **zugegriffen** werden
- ▶ `SlimeBlob.InnerBlobling.getParentSize()`

```
public int getParentSize() {  
    return SlimeBlob.this.size;  
}
```

- ▶ `InnerBlob` kann mit `SlimeBlob.this` auf **alle Attribute** der umschließenden Instanz zugreifen
- ▶ Somit: `SlimeBlob.this` **entspricht** `Blobling.parent` von vorher
- ▶ **Hinweis:** Wäre `InnerBlobling` und Konstruktor **public**, so wäre es Instanziierung von **außen** möglich über:

```
SlimeBlob.InnerBlobling b =  
    slimeBlobParent.new InnerBlobling(...);
```

281

## Hinweise zu inneren Typen

- ▶ Gleiche **Regeln** bezüglich Sichtbarkeit wie bei **statischen geschachtelten Typen**
  - ▶ **public**, Paket-sichtbar, **protected**, **private**
- ▶ Auch **enum**, **interface** können **schachteln** und **geschachtelt** werden (geschachtelte **enums** sind implizit **static**)
- ▶ **Anwendung** innerer Typen
  - ▶ Prinzipiell wie bei **statischen geschachtelten Typen**
  - ▶ Immer wenn innerer Typ an **umschließende Instanz** gebunden
  - ▶ Z.B. für [↗ Iterator](#) (später)

282

## Geschachtelte Typen

### Lokale Klassen

## Lokale Klassen

- ▶ Klassen können sogar **innerhalb von Methoden** deklariert werden

```
public void doSomething(){  
    class LocalClass{  
        private int innerAttribute;  
        LocalClass() { }  
        public void innerMethod(){ }  
    }  
    var l = new LocalClass();  
    l.innerMethod();  
}
```

- ▶ **Besonderheiten**
  - ▶ Dürfen **keine Sichtbarkeit** definieren
  - ▶ Sind nur in **deklarierender Methode** sichtbar
  - ▶ Zugriff auf **final** lokale Variablen
  - ▶ Attribute der **umschließenden Klasse** (vgl. **inneren Klassen**)

## SlimeBlob.localDivide() I

Wie sieht das für unsere SlimeBlob-Klasse aus?

```
102 public SlimeBlob[] localDivide(){
104     // Klassendefinition
105     class LocalBlobling extends SlimeBlob{
107         private LocalBlobling(String name,
108             int x, int y, int health, int attackPower,
109             int size, SlimeColor color){
110             super(name,x,y,health,attackPower,size,color);
111         }
113         public SlimeBlob getParent() { return SlimeBlob.this; }
114     }
116     // Verwendung
117     LocalBlobling child1 =
118         new LocalBlobling(getName()+"ling 1", getAttackPower(),
119             getX(), getY(), getHealth(), size/2, color);
120     LocalBlobling child2 =
```

285

## SlimeBlob.localDivide() II

```
121         new LocalBlobling(getName()+"ling 2", getAttackPower(),
122             getX(), getY(), getHealth(), size/2, color);
124     return new SlimeBlob[] { child1, child2 };
125 }
```

game/SlimeBlob.java

### ► Hinweise

- Zugriff auf umschließende Instanz hier über **SlimeBlob.this**
- LocalBlobling **nur in Methode** sichtbar (nach außen SlimeBlob)

286

## Hinweise zu lokalen Klassen

- ▶ Sollte nur für **kurze Klassen** verwendet werden
- ▶ **Sonst lieber**
  - ▶ Statischer geschachtelter Typ
  - ▶ Innerer Typ
- ▶ **Erfahrung aus Praxis**
  - ✗ Mischung von **Methodenrumpf-Semantik** und **Deklarations-Semantik**
  - ▶ Eher selten anzutreffen

287

## Inhalt

**Geschachtelte Typen**  
Anonyme Klassen

288



## Anonyme Klassen

- ▶ Anonyme Klassen sind unbenannte lokale Klassen

```
public void doSomething(){
    Base b = new Base() {
        @Override public void baseMethod(){
            // do something else
        }
    };
    b.baseMethod();
}
```

- ▶ Objekttyp von b erbt von Base und überschreibt Methode
- ▶ Besonderheiten
  - ▶ Keine Sichtbarkeit und kein Name
  - ▶ Nur ein Basistyp (Klasse oder interface)
  - ▶ Zugriff auf Elemente von Base, final lokale Variablen und umschließende Klasse
  - ▶ Nur Default-Konstruktor über Initializer (s. unten)

289

### SlimeBlob.anonymousDivide |

Wie sieht das für unsere SlimeBlob-Klasse aus?

```
129 public SlimeBlob[] anonymousDivide(){
131     final int outerParentSize = size;
133     // Kinder erstellen
134     SlimeBlob child1 = new SlimeBlob(getName(), getX(), getY(),
135         getHealth(), getAttackPower(), size/2, color){
137         public SlimeBlob getParent(){
138             return SlimeBlob.this;
139         }
140     };
142     SlimeBlob child2 = new SlimeBlob(getName(), getX(), getY(),
143         getHealth(), getAttackPower(), size/2, color){
144         private int parentSize;
146         { // Initializer
147             this.parentSize = outerParentSize;
148         }
149     };
150 }
```

290


## SlimeBlob.anonymousDivide II

```
150     public int getParentSize(){
151         return parentSize;
152     }
153 };
154 return new SlimeBlob[] { child1, child2 };
155 }
```

game/SlimeBlob.java

291

## SlimeBlob.anonymousDivide

```
38  runAnonymousBlobsExample
39 out.println(slimeBlob);
41 SlimeBlob[] children = slimeBlob.anonymousDivide();
42 out.println(children[0]);
43 out.println(children[1]);
45 out.println(children[0].getClass().getName());
46 out.println(children[1].getClass().getName());
```

NestedTypesExamples.java

```
: name="Blob", health=100, x=0, y=1, attackPower=20, size=30, color=RED
: name="Blob", health=100, x=0, y=1, attackPower=20, size=30, color=RED
de.hawlandshut.java1.oop.game.SlimeBlob$1
de.hawlandshut.java1.oop.game.SlimeBlob$2
```

292

- **Beobachtung:** Für jedes Kind wird eine **eigene** anonyme Klasse definiert!

```
de.hawlandshut.java1.oop.game.SlimeBlob$1  
de.hawlandshut.java1.oop.game.SlimeBlob$2
```

```
child1.getClass() != child2.getClass()
```

- Methode getParentSize ist **nur** in SlimeBlob.anonymousDivide **zugreifbar**

```
child1.getParentSize();
```


## Anonyme Klassen für Interfaces

- **Historisch:** anonyme Klassen für **lokale Implementierung** von **interfaces**
- **Vor allem** für Event-Interfaces wie „**Button gedrückt**“
- **Mittlerweile** abgelöst durch **Lambda-Ausdrücke** (Programmieren III)
- Beispiel für **interface** `game/Container`

```
Container c = new Container(){  
    @Override public int unitsLeft(){  
        return 1;  
    }  
    @Override public int maximumUnits(){  
        return 10;  
    }  
};  
out.println(c.unitsLeft()); // 1  
out.println(c.maximunUnits()); // 10
```

## Beispiel: Anonyme Klasse für Interface I

Längeres **Beispiel** implementiert Consumable

```
51  runAnonymousConsumableExample  
52 public static void anonymousConsumableExample() {  
53     Consumable soup = new Consumable(){  
54         private int spoonsLeft = 100;  
55         @Override public int unitsLeft(){  
56             return spoonsLeft;  
57         }  
58         @Override public void consumeOne(){  
59             if (spoonsLeft > 0)  
60                 spoonsLeft--;  
61         }  
62         @Override public Consumable split(int n) {  
63             throw new UnsupportedOperationException("nah");  
64         }  
65     };  
66     out.printf("Spoons left: %d\n", soup.unitsLeft());  
68 }
```

295

## Beispiel: Anonyme Klasse für Interface II

```
69     soup.consumeOne();  
70     out.printf("Spoons left: %d\n", soup.unitsLeft());  
71 }
```

 NestedTypesExamples.java

```
Spoons left: 100  
Spoons left: 99
```

► Beispiel zeigt: Anonyme Klassen können auch eine **Zustand** halten

296

## Hinweise: Anonyme Klasse

- ▶ Prinzipiell wie lokale Klassen
- ▶ Unterschiede
  - ▶ Nur Default-Konstruktor über Initializer
  - ▶ Kein Name
  - ▶ Nur ein Basistyp (**interface** oder Klasse)
- ▶ Verwendung
  - ▶ Kurze Implementierungen
  - ▶ Meist für **interfaces**
- ▶ Seit **Lambdas** außer Mode geraten

297

## Inhalt

Geschachtelte Typen  
Zusammenfassung

298

## Zusammenfassung

	Kontext	Sichtb.	sieht	Typ	new
Top-Level	Global	~, +	<b>this</b>	*	<b>new</b> C()
<b>static</b>	Top-Level C	*	<b>this</b>	*	<b>new</b> C.Nested()
innere	Objekt o	*	<b>this</b> , C. <b>this</b>	*	o. <b>new</b> Nested()
lokal	Methode	—	C. <b>this</b> , lok. Var.	<b>class</b>	<b>new</b> Nested()
anonym	Methode	—	C. <b>this</b> , lok. Var.	<b>class</b>	<b>new</b> Base(){}

\* steht für

- ▶ **public**, Paket-sichtbar, **protected**, **private** bei Sichtbarkeit
- ▶ **class**, **interface**, **enum** bei Typ

~(Paket-sichtbar), + (**public**)