Objektorientierte Programmierung mit Java – Woche 4

openHPI-Java-Team

Hasso-Plattner-Institut

Wiederholung Woche 3

openHPI-Java-Team

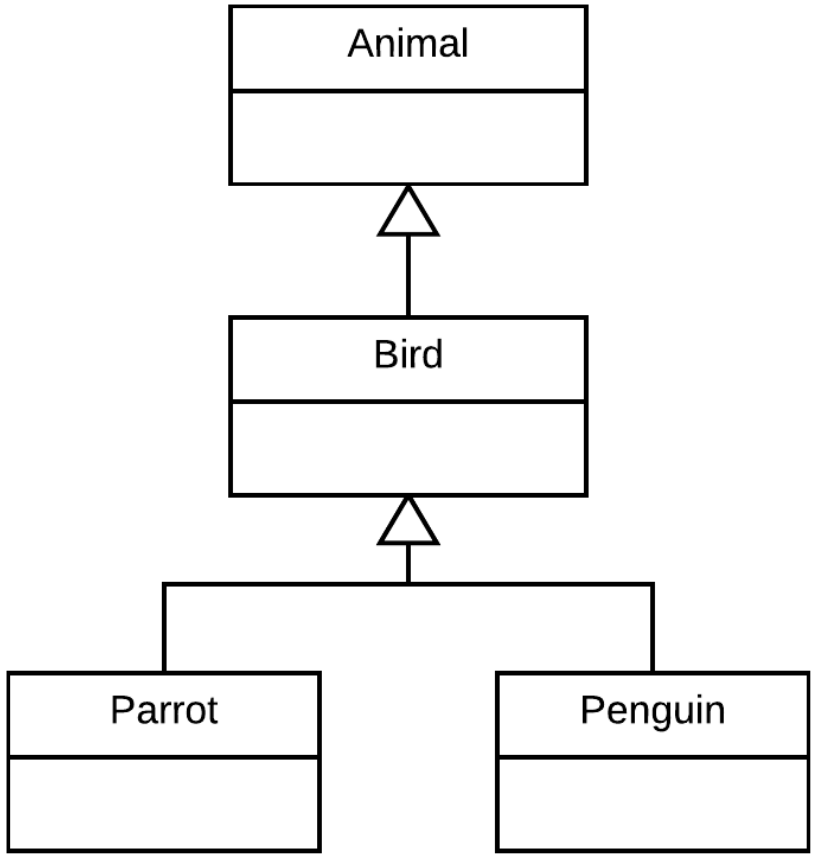
Hasso-Plattner-Institut

Vererbung

- Ein Papagei **ist ein** Vogel
- Ein Pinguin **ist ein** Vogel
- Ein Vogel **ist ein** Tier

- *Is-a Beziehung* in einem UML Klassendiagramm
  - A parrot **is a** bird
  - A parrot **is an** animal

- Die Klasse Parrot erhält transitiv alle Methoden und Attribute der Klasse Animal



Vererbung

```java
class Story {
   public static void main(String[] args) {
        Parrot paco = new Parrot();
        paco.name = "Paco";
   }
}
```

```java
class Bird {
   String name;
   // ...
}
```

```java
class Parrot extends Bird {
}
```

- Subklassen erben implizit Attribute und Methoden der Superklasse
- Subklassen können eine Superklasse um zusätzliche Attribute und Methoden erweitern
- Subklassen können Methoden von Superklassen überschreiben

Sichtbarkeiten: Schlüsselwörter

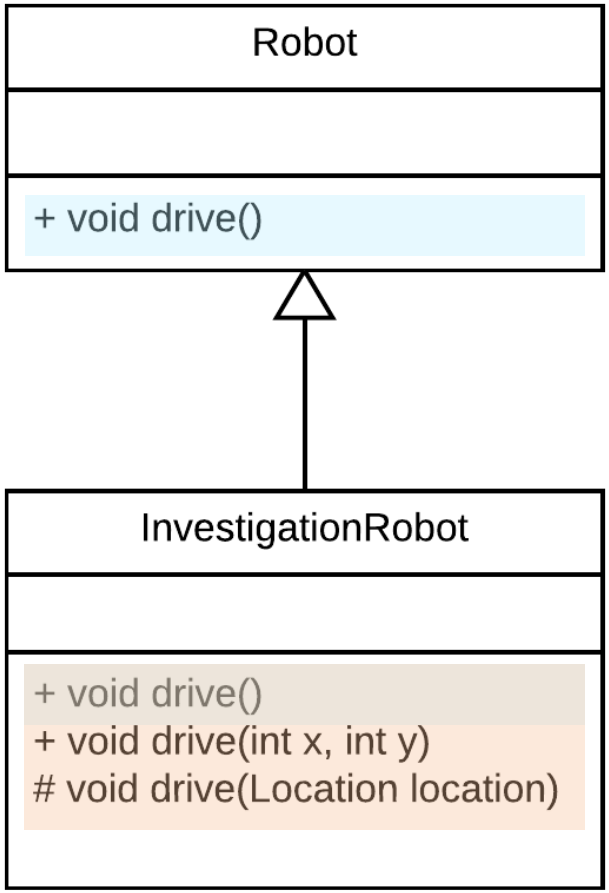
Sichtbarkeit	Bedeutung
`public`	uneingeschränkter Zugriff
`protected`	Zugriff aus der eigenen Klasse und deren Subklassen
default (keine Sichtbarkeit angegeben)	Zugriff für Klassen des Packages
`private`	Kein Zugriff für andere Klassen, nur für Objekte der Klasse

Kapselung: Getter und Setter

```
1  public class Parrot {
2     private String name;
3
4     public String getName() {
5         return this.name;
6     }
7     public void setName(String value) {
8         this.name = value;
9     }
10 }
```

- Konvention: Alle Attribute `private`
- Zugriff optional über `public` Getter/Setter möglich
  - Ermöglichen es Entwicklern anderen Objekten Lese- bzw. Schreibrechte auf Attribute einzuräumen
  - Vollständige Kontrolle des Zugriffs

Override vs. Overload



Überschreiben (Override)

- Methode in Subklasse überschreibt Methode in Superklasse

Überladen (Overload)

- Innerhalb einer Klasse

Abstrakte Klassen und Methoden

```
1 public abstract class Animal {
2    String name;
3
4    public abstract void eat();
5 }
```

Abstrakte Klassen

- Werden durch das Schlüsselwort `abstract` angelegt
- Können nicht instanziiert werden

Abstrakte Methoden

- Werden durch das Schlüsselwort `abstract` angelegt
- Haben keinen Methodenrumpf und enden mit einem Semikolon
- Müssen von der ersten konkreten Subklasse überschrieben werden (und damit definiert)

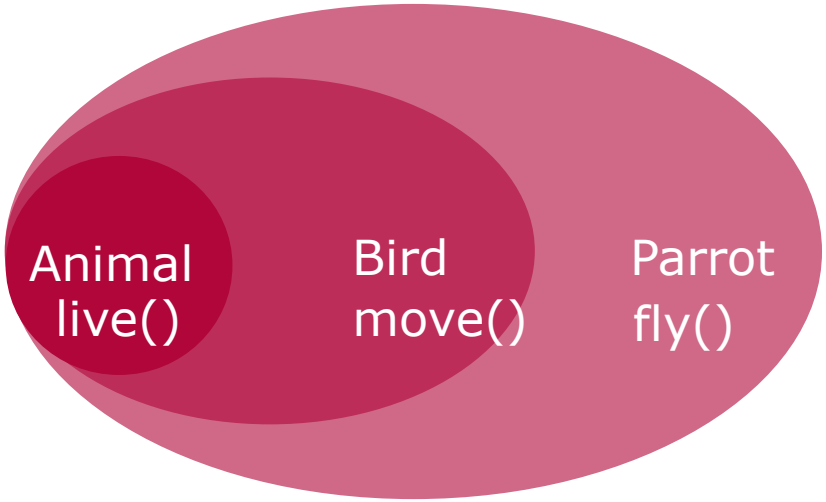
Polymorphie

```
  [...]
1    Bird penguin = new Penguin();
2    Bird parrot = new Parrot();
3
4    Bird[] array = new Bird[2];
5    array[0] = parrot;
6    array[1] = penguin;
7
8    for (int i = 0; i < array.length; i++) {
9        array[i].move();
10   }
11 [...]
```

Ausgabe:

```
Ich fliege
Ich schwimme
```

Polymorphie



- `Animal paco = new Parrot(); paco.live(); ~~paco.move();~~ ~~paco.fly();`
- `Bird paco = new Parrot(); paco.live(); paco.move(); ~~paco.fly();`
- `Parrot paco = new Parrot(); paco.live(); paco.move(); paco.fly();`

- Polymorphie = “Vielgestaltigkeit”
- Man kann ein Objekt als Instanz mehrerer Klassen betrachten

Objektdatentypen

openHPI-Java-Team

Hasso-Plattner-Institut

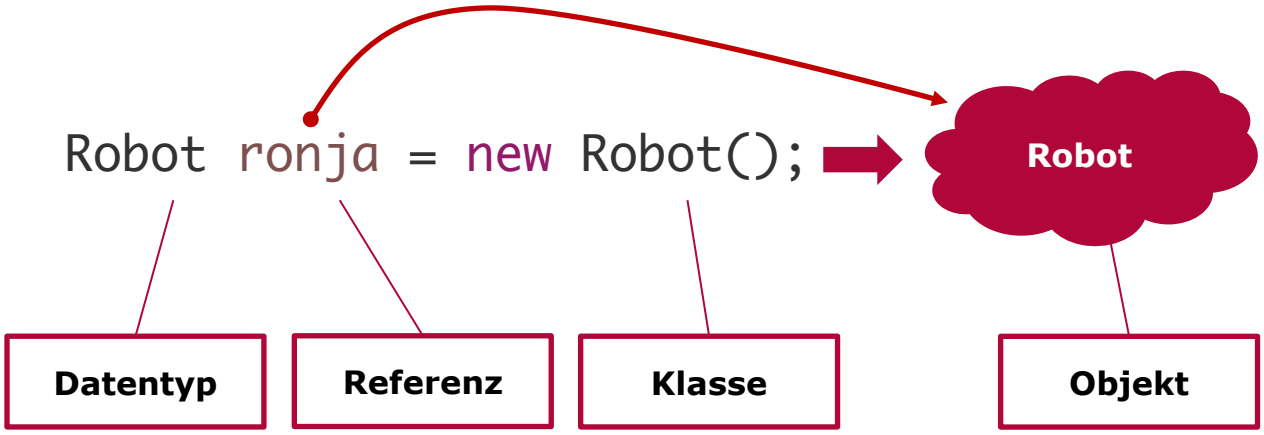
Primitive Datentypen (Wiederholung Woche 1)

- Erkennbar: kleingeschrieben, Schlüsselwörter
- Keine Methoden oder Attribute

Name	Art	Beispiele
`char`	Buchstabe, Zeichen	`char country = 'd';` `char cedille = 'ç';`
`int`	Ganzzahl	`int age = 2;` `int truth = -42;`
`double`	Kommazahl	`double speed = 98.7;`
`boolean`	Wahrheitswert	`boolean isGreen = true;`

Objektdatentypen

- **Jede Klasse ist auch ein (Objekt-)Datentyp**
  - Bereits bekannt: `String` → ist ein Objektdatentyp in der Java API
  - https://docs.oracle.com/javase/8/docs/api/
- Robot ist ein von uns selbstdefinierter Objektdatentyp



equals und toString (1/3)

- Jede Klasse hat standardmäßig u.a. die zwei Methoden
  - `equals(Object x)`
    - Vergleicht zwei Objekte
    - Standard: Vergleich der Speicheradressen der Objekte
    - Für eigene Klassen können andere Eigenschaften maßgeblich sein
  - `toString()`
    - Repräsentiert ein Objekt als String
    - Standard: *Klassenname@Speicheradresse*
    - Rückgabe sollte an die jeweilige Klasse angepasst werden

➔ **Ändern für eigene Klassen sinnvoll**

equals und toString (2/2)

```java
public class Parrot /* extends Object */ {
   private String name;
   private int age;

   @Override
   public boolean equals(Object o) {
       // [...] sicherstellen dass o vom richtigen Typ ist
       // Deep Dive
       return (o != null) &&
              this.name.equals(((Parrot) o).name) &&
              this.age == ((Parrot) o).age;
   }

   [...]
}
```

equals und toString (2/3)

Hasso Plattner Institut

```java
public class Parrot /* extends Object */ {
    private String name;
    private int age;

    [...]

    @Override
    public String toString() {
        String newline = System.getProperty("line.separator");
        String output = "Parrot: " + newline;
        output += "Name: " + this.name + newline;
        output += "Age: " + this.age + newline;
        return output ;
    }
}
```

Wrapper

- Jeder primitive Datentyp hat einen passenden Objektdatentyp
- Bieten zusätzlich:
  - Methoden, die spezifisch für diesen Datentyp sind → z.B. `Integer.parseInt(String s)` zum Umwandeln eines Strings
  - Konstanten, z.B. `Integer.MAX_VALUE` → 2147483647

Primitiver Datentyp	Zugehöriger Objektdatentyp
`int`	`Integer`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

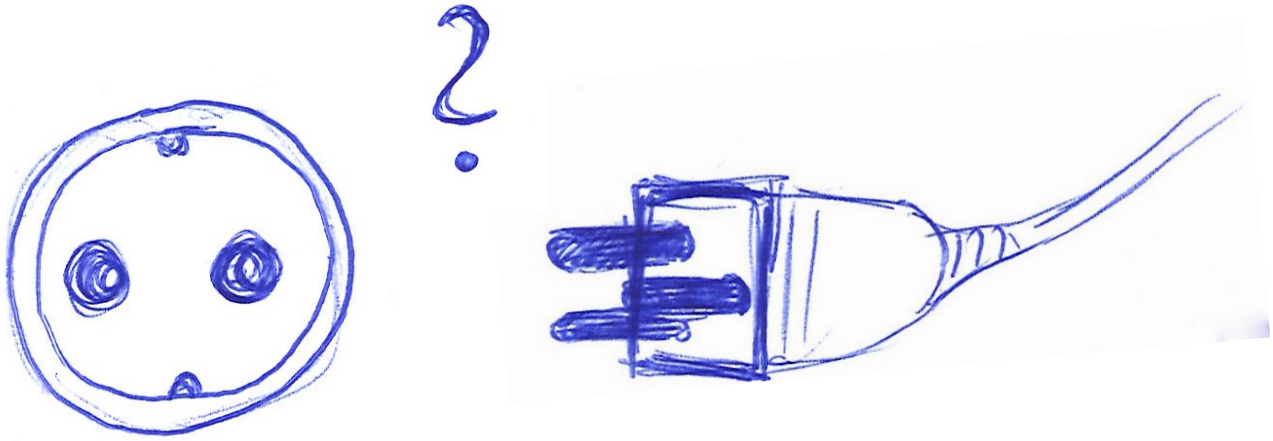
Type Casting

openHPI-Java-Team

Hasso-Plattner-Institut

Was tun - wenn mein Datentyp gerade nicht passt?

- Parameter und Rückgabewert haben vordefinierte Datentypen
- Manchmal liegt gerade ein anderer Typ vor als der den man braucht
- → temporäre Umwandlung (Type Casting)
- Funktioniert nur mit **kompatiblen** Typen



Implizites Type Casting

```
1 int x = 2;
2 double y = x;
3 // y speichert nun den Wert 2.0
4 // gleiches Ergebnis bei double y = 2;
```

- Nur möglich, wenn keine Information verloren geht

  → 2 wird zu 2.0
- Geschieht implizit ohne weitere Eingriffe durch den Entwickler

  → Keine spezielle Syntax nötig

Explizites Type Casting
Numerische Typen (1/2)

```
1 double y = 2.5;
2 int x = y;          <-- FEHLER
3 int x = (int) y;
4 // y wird abgeschnitten nach dem Komma
5 // x enthält nun also 2
```

- Compiler erzeugt Fehler bei **potentiellem Informationsverlust**
- Type Casting kann erzwungen werden
  - Zieltyp in runden Klammern vor den Bezeichner schreiben
  - Hallo Compiler! Ich übernehme hier mal selbst die Verantwortung!
- Funktioniert nur zwischen kompatiblen Datentypen

Explizites Type Casting klassischer Anwendungsfall (2/2)

```
1 double z = 1 / 2.0;
2
3 int x = 1;
4 int y = 2;
5
6 z = x / y;
7 z = x / (double) y;
```

z == 0.5

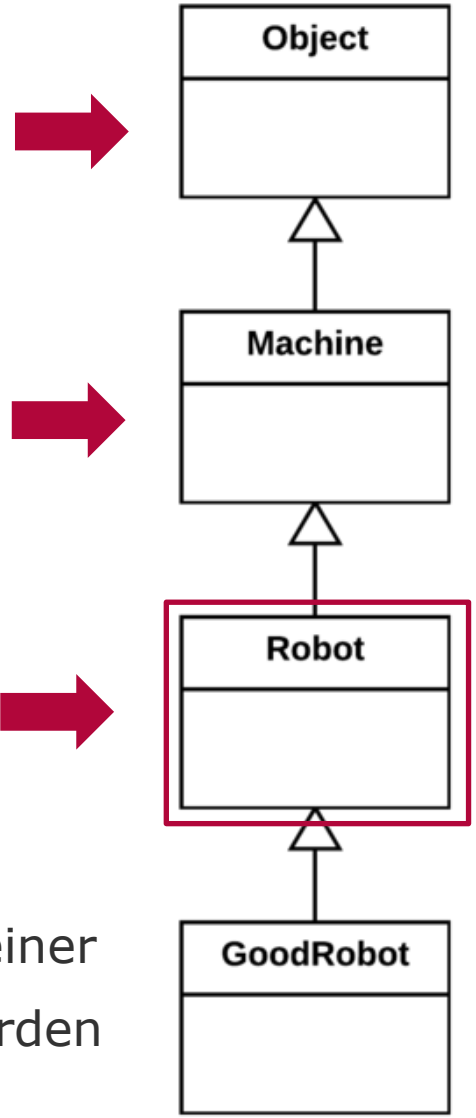
x/y == 0 ➔ z == 0.0

z == 0.5

- Division ergibt für Ganzzahlen keine Nachkommastellen
- → alles was zwischen 0 und 1 ist wird als 0.0 ausgegeben
- Erst wenn Zähler Kommazahl ist, wird Ergebnis mit Nachkommastellen gespeichert

Explizites Type Casting Objektdatentypen

```
1 Object robbi = new Robot();
2
3 Machine m = (Machine) robbi;
4
5 Robot richard = (Robot) m;
6
7 Robot randy = (Robot) robbi;
8
9 GoodRobot rolf = (GoodRobot) robbi;
10
```



- Jedes Objekt kann in jeden Datentyp innerhalb seiner (**linearen**) Vererbungshierachie umgewandelt werden

Explizites Typecasting
Objektdatentypen (2/2)

```
1 Robot aRobot = new Robot();
2
3
4 String s = (String) aRobot;
5 /* wirft ClassCastException, da für das Object
6 aRobot nur die Eigenschaften von Robot
7 vorhanden sind und nicht von String */
```

FEHLER

Parsing
String → Zahl

Hasso Plattner Institut

```java
double x = Double.parseDouble("1.234");
int y = Integer.parseInt("1234");
boolean b = Boolean.parseBoolean("true");
// nutzt bereits implementierte Methoden
```

- Statt Casting
- Bereitgestellte Methoden durch Java Wrapper-Klassen
- Wenn Zahl als String bekannt (z.B. "1.234")
- Funktioniert nicht für Strings, die nicht-numerisch sind
- → `int z = Integer.parseInt("Ich fliege");` wirft FEHLER
- → NumberFormatException

Collections

openHPI-Java-Team

Hasso-Plattner-Institut

Collections (Sammlungen)

- Organisieren Objekte beliebiger (Objekt-)Datentypen
- Collections sind selbst auch Objektdatentypen
  → besitzen Methoden und Attribute
- Verschiedene Collections haben unterschiedliche Eigenschaften:
  - Array
  - List
  - Map
  - Set

- Elementdatentyp in spitzen Klammern <>:
  - Definiert den Typ der in der Collection erlaubten Elemente
  - Bsp: `ArrayList<String> words = new ArrayList<>();`

Wiederholung Primitive Arrays

```
1  int[] numbers = new int[5];
2  numbers[0] = 6;
3  numbers[1] = 33;
4  numbers[2] = 9;
5  numbers[3] = 0;
6  numbers[4] = 503;
```

Index	Wert
0	6
1	33
2	9
3	0
4	503

- Primitive Arrays können ihre Größe **nicht** verändern → zusätzliche Elemente können nur unter sehr großem Aufwand angefügt werden
- `length` gibt mir nur die Größe, aber nicht die Anzahl der real existierenden Elemente
- Soll ein Element vorn eingefügt werden, müssen die folgenden Elemente manuell verschoben werden → Was geschieht mit dem letzten Element?

Array (1/3)

- Array mit veränderlicher Größe
- Schneller Zugriff auf Elemente an beliebiger Position
- Ist ein Array voll wird automatisch ein neues angehängt

Index	Wert
0	6
1	33
2	9
3	0
4	503

5	22
6	31
7	333
8	6
9	5603

ArrayList (2/3)
ein Codebespiel

```
1 ArrayList<String> list = new ArrayList<>();
2 list.add("Hallo");
3 list.add("Huhu");
4 list.size();
5 list.get(0);
6 list.remove(0);
7 list.get(0);
```

2

Hallo

Huhu

Neue Elemente anfügen

Anzahl der Elemente

Erstes Element ausgeben

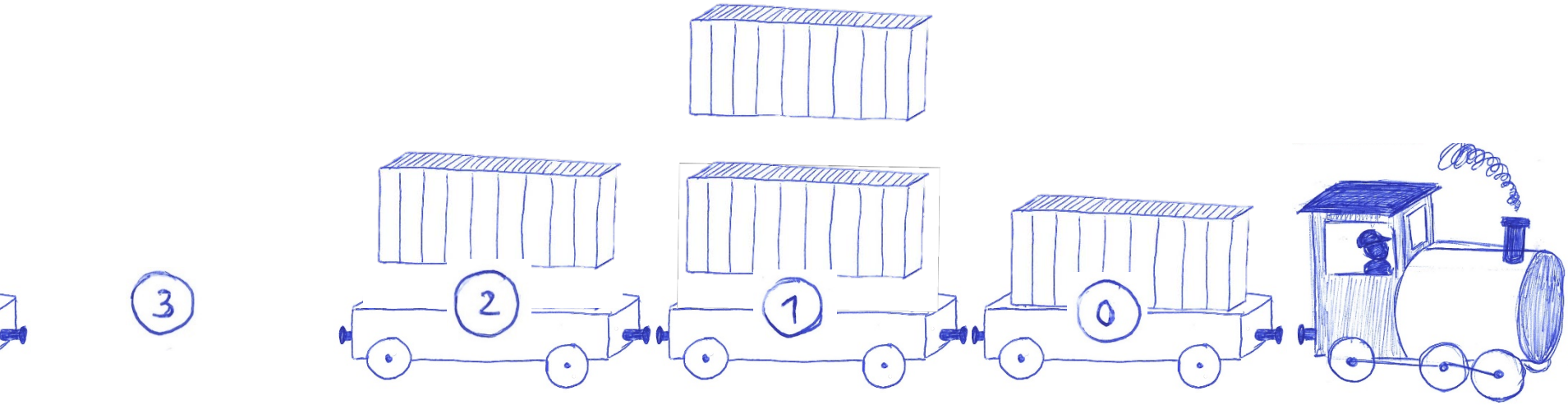
Erstes Element löschen

Erstes Element ausgeben

ArrayList (3/3)

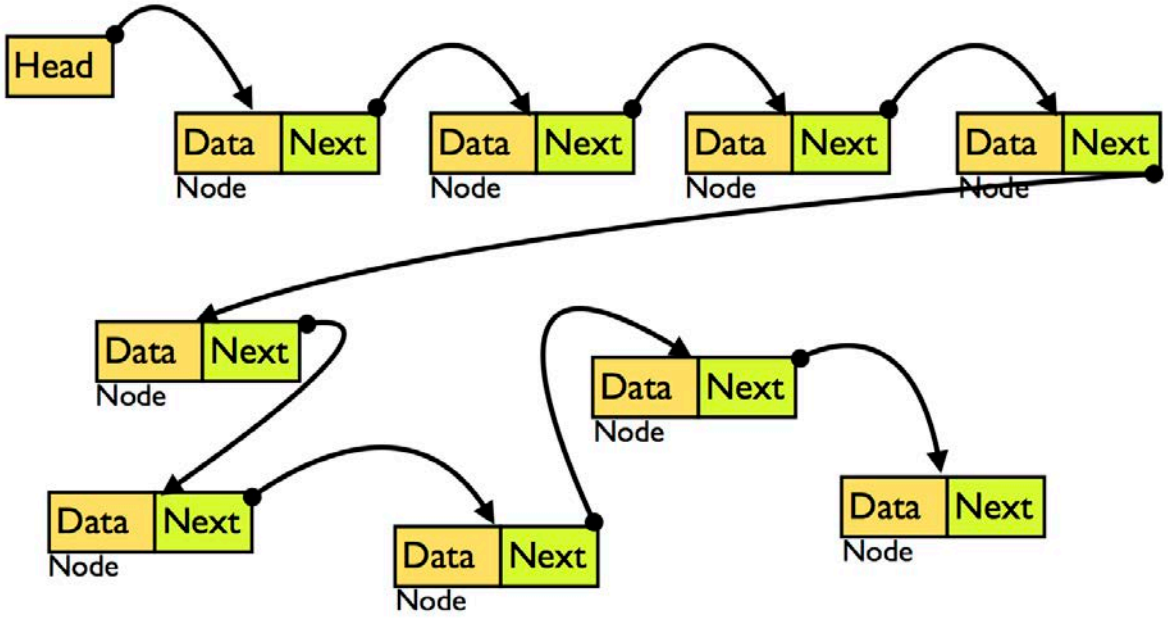
- Elemente vorn einfügen kann „teuer“ werden
- Nachfolgende Elemente werden automatisch nach hinten verschoben

➔ Dieses Problem wurde sozusagen automatisiert aber nicht behoben



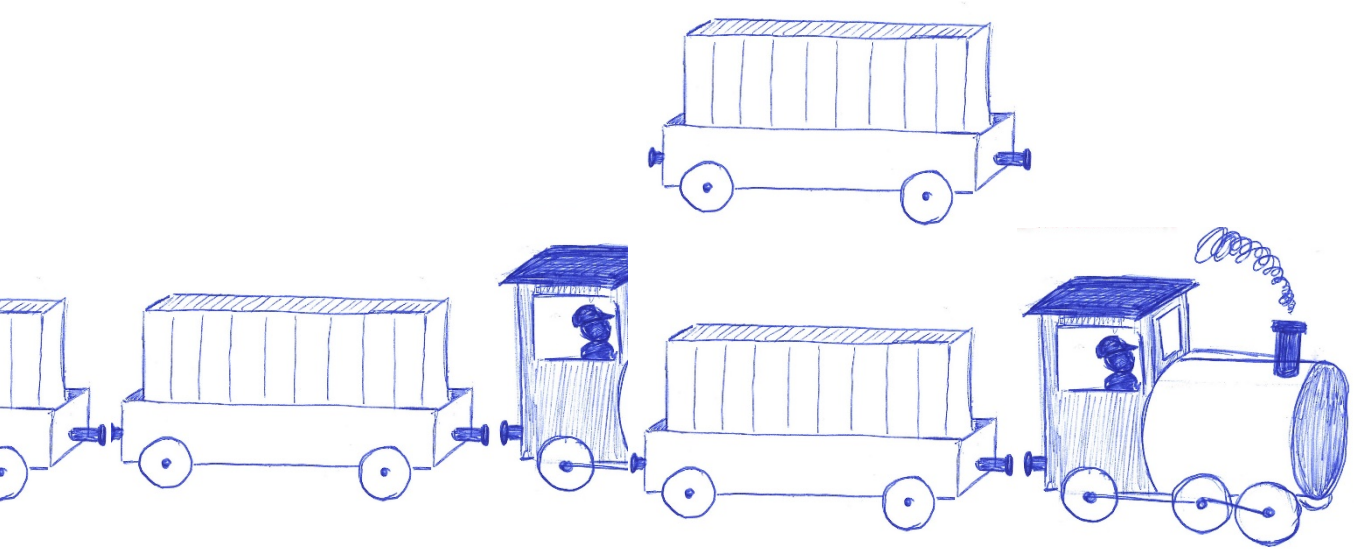
List (1/3)

- Verkettet **Nodes** miteinander
- Nodes enthalten **Referenz auf folgenden Node** (Next) und Referenz auf das enthaltene **Datenobjekt** (Data)
- Nachteil: Kein Random Access → um zu einem Element zu gelangen muss über alle vorhergehenden Elemente iteriert werden



List (2/3)

- Vorteil: Einfaches Einfügen und Entfernen von Elementen auch am Anfang oder in der Mitte der Liste
- Node einfügen/löschen → Referenz auf nächsten Node „umbiegen“



LinkedList (3/3)
ein Codebespiel

```
1 LinkedList<String> list = new LinkedList<>();
2 list.add("Ronja");
3 list.add("Robin");
4
5 list.size();
6 list.getFirst();
7 list.removeFirst();
```

Neue Elemente einfügen

Anzahl der Elemente

Erstes Element ausgeben

Erstes Element löschen

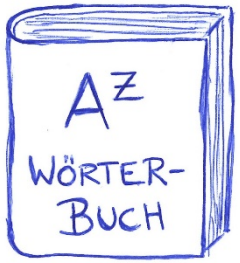
Spezielle Listen:

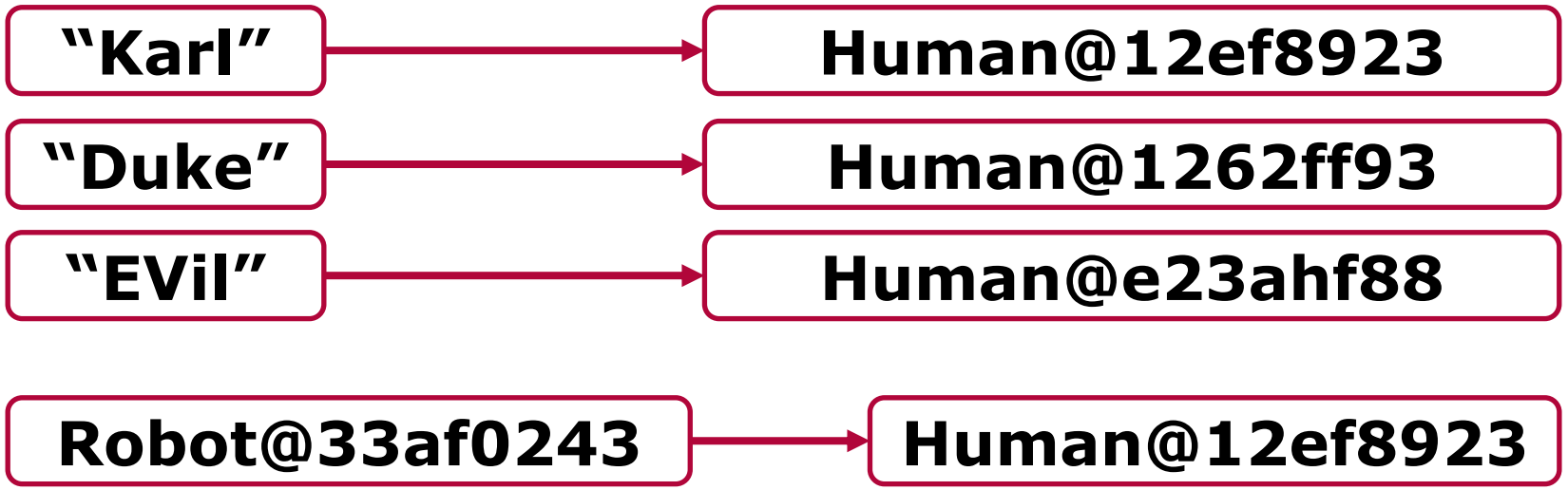
- Queue: First In First Out (FIFO)
  - Schlange

- Stack: Last In First Out (LIFO)
  - Stapel

Map (1/2)

- Jedem Wert wird ein Schlüssel zugeordnet (KEY-VALUE Paar)
  - Schlüssel (KEY) ermöglicht Zugriff auf Wert (VALUE).
- Im Gegensatz zum Array ist der Schlüssel (in der Regel) nicht-numerisch
- Elemente sind „unsortiert“





HashMap (2/2)

```
1 HashMap<Customer, Robot> orders = new HashMap<>();
2 orders.put(duke, new Robot("Ronja"));
3 orders.put(eVil, new Robot("Robur"));
4 orders.size();
5 orders.get(duke);
6
7 orders.remove(eVil);
8
9 orders.containsKey(eVil);
```

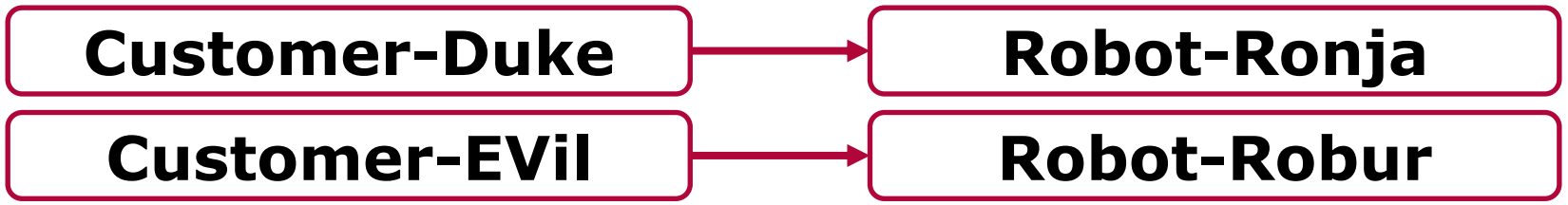
Neue Elemente einfügen

Anzahl der Elemente

Element mit dem KEY duke ausgeben

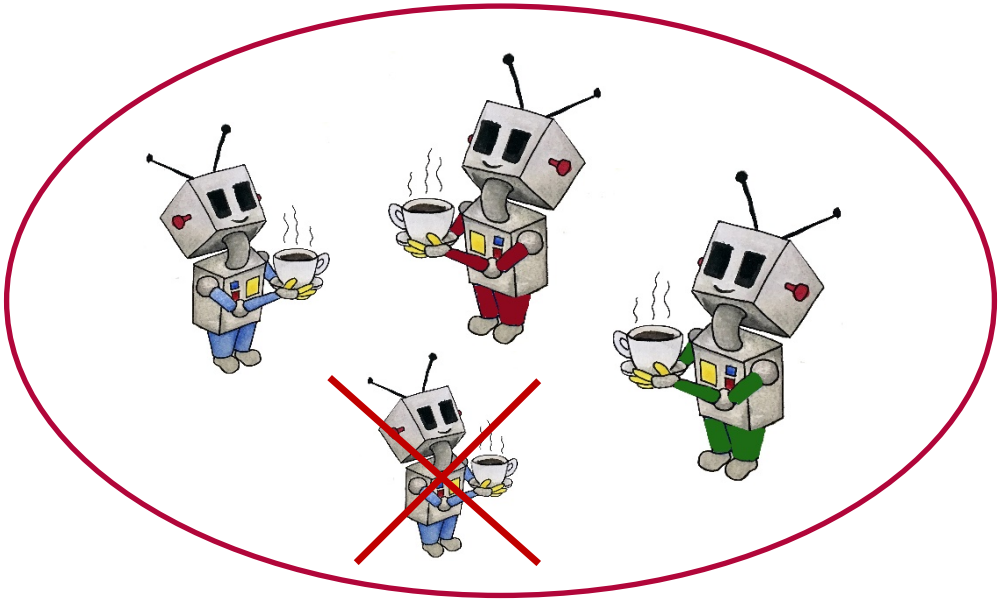
Element mit dem KEY eVil löschen

Prüfen ob ein KEY existiert



Set (1/1)

- Menge
- Jedes Element kann nur **einmal** enthalten sein ➔ keine Duplikate



- Verschiedene Implementierungen in Java, z. Bsp.:
  - HashSet: unsortiert

Collections in der Java API

- Collections sind in der Regel die bessere Wahl als primitive Arrays
- Automatisierung vieler House-Keeping Funktionen
- Collections können nur Objektdatentypen beinhalten
  - Wrapper-Klassen für primitive Datentypen
- **ArrayList** – für Random Access mittels durchgängiger numerischer Schlüssel
- **LinkedList** – wenn häufig Elemente im „vorderen" Bereich eingefügt werden müssen
- **HashMap** – wenn der Zugriff über nicht-numerische Schlüssel wichtig ist
- **HashSet** – wenn sicher gestellt werden soll, dass es keine Duplikate gibt

Collections in der Java API

- JavaAPI
  https://docs.oracle.com/javase/8/docs/api/
- ArrayList
  https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
- LinkedList
  https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html
- HashMap
  https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html
- HashSet
  https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html

Foreach-Schleifen

openHPI-Java-Team

Hasso-Plattner-Institut


```
                          0       1       2
1  String[] words = {"Hallo", "Huhu", "Na?"};
2
3  for( int i = 0; i < words.length; i++){
4          System.out.println(words[i]);
5  }
```

- Anzahl der Elemente eines Arrays mit <Arrayname>.length abfragen

Ausgabe:

Hallo ← Ausgabe für i = 0

Huhu ← Ausgabe für i = 1

Na? ← Ausgabe für i = 2

foreach-Schleifen (1/3)

- Bei for-Schleife langer Schleifenkopf nötig, um alle Elemente zu betrachten:
  - `for( int i = 0; i < array.length; i++){ ... }`

- Java hat auch verkürzte Form
  → foreach-Schleife
  - `for( String s : words ){ ... }`
  - Bedeutet „Für jeden `String s` in dem Array `words` mach...“
  - Zählvariable nicht länger benötigt
  - Anwendbar auf Arrays und Collections

Achtung! Beim Durchlaufen können keine Werte gelöscht oder eingefügt werden (Lösung: Iteratoren)

foreach-Schleifen (2/3)

```
1  String[] words = {"Hallo", "Huhu", "Na?"};
2
3  for( String s : words ){
4          System.out.println(s);
5  }
```

Ausgabe:

Hallo ← 1. Element

Huhu ← 2. Element

Na? ← 3. Element

foreach-Schleifen (3/3)

```
1  ArrayList<String> words = new ArrayList<>();
2
3  words.add("Hallo");
4  words.add("Huhu");
5  words.add("Na?");
6
7  for( String s : words ){
8          System.out.println(s);
9  }
```

Ausgabe:

Hallo ← 1. Element

Huhu ← 2. Element

Na? ← 3. Element

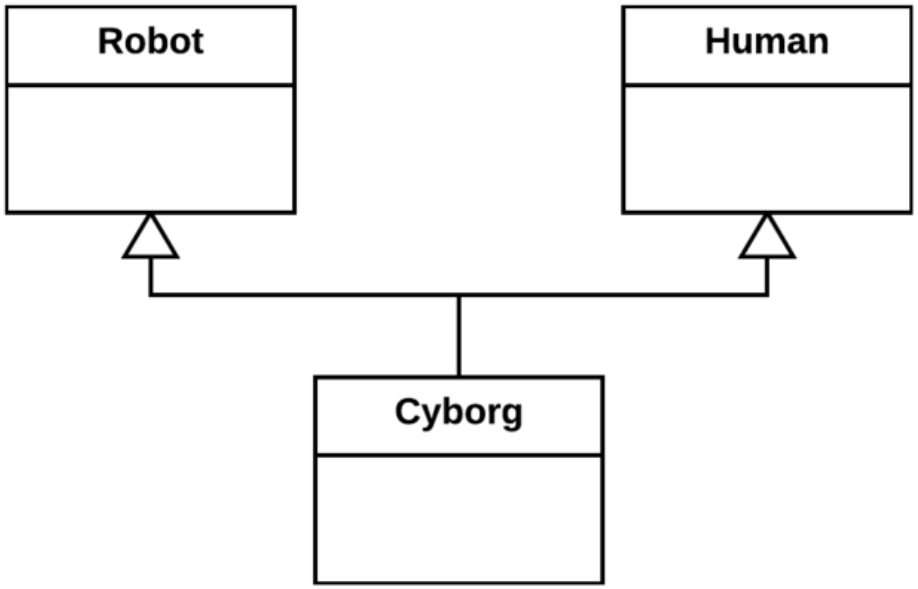
Interfaces

openHPI-Java-Team

Hasso-Plattner-Institut

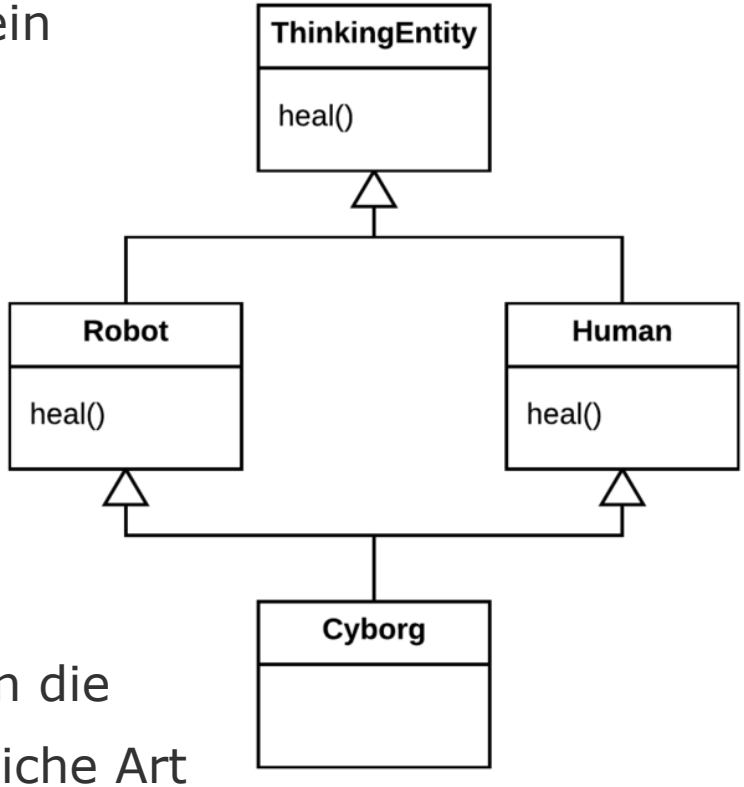
Prinzip der Mehrfachvererbung

- Eine Subklasse erbt aus (potentiell) verschiedenen Vererbungshierarchien
- Erscheint manchmal auf den ersten Blick als vorteilhaft



Das Diamond-Problem

- ABER: Mehrfachvererbung verursacht ein großes Problem:



- Die Klassen `Human` und `Robot` definieren die Methode `heal()` auf unterschiedliche Art
- **Welche der Implementierungen erbt der `Cyborg`, wenn `heal()` nicht neu definiert wird?**

→ **Java unterstützt keine Mehrfachvererbung**

Die Lösung: Interfaces

- Warum wollten wir Mehrfachvererbung haben?

➔ Es sollte sichergestellt werden, dass sowohl die Methoden der Klasse `Human` als auch die Methoden der Klasse `Robot` vorhanden sind.

- Interfaces
  - Stellen einen „Vertrag" zur Verfügung, den die implementierende Klasse einhalten muss.
  - Jeder Entwickler kann sich darauf verlassen, dass jede Klasse die ein Interface implementiert auch deren Methoden implementiert.
  - Sind sozusagen 100% abstrakte Klassen ➔ Sie beinhalten ausschließlich abstrakte Methoden

Die Lösung: Interfaces

- Syntax Definition:

```
interface Healable {  //Konvention: -able
    /* public abstract*/ int heal();
    /*Alle Methoden in Interfaces müssen public und
    abstract sein, daher kann man die Modifikatoren
    weglassen.*/
}
```

- Syntax Nutzung:
- `class Cyborg implements Healable { ... }`
- `class <Klassenbezeichner> implements <InterfaceBezeichner>`
- Eine **nicht-abstrakte** Klasse die ein Interface implementiert, muss alle Methoden die im Interface deklariert wurden implementieren.

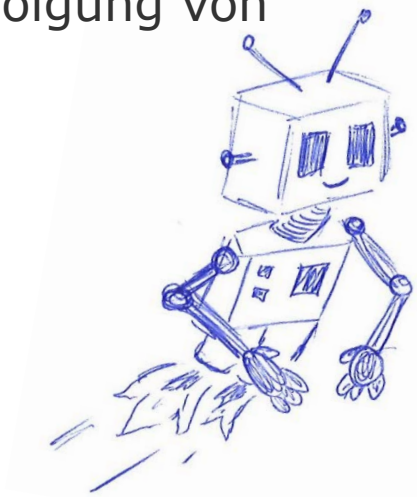
Wozu brauche ich das?
Ein Beispiel (1/4)

- Mal angenommen:
  - Paco schafft es sich aus der Zentrale von Eike Vil zu befreien.
  - Ronja hat kürzlich ein Upgrade erhalten und kann nun auch fliegen.
  - Eike Vil hat einen SuperVillain-Umhang im Schrank und kann damit auch fliegen.

➔ Alle können sich eine Verfolgungsjagd in der Luft liefern

Wozu brauche ich das?
Ein Beispiel (2/4)

- Paco flieht.
- Ronja versucht Paco einzuholen und mit ihm zu Detektiv Duke zu fliegen.
- Eike Vil nimmt die Verfolgung von Paco und Ronja auf.



Wozu brauche ich das?
Ein Beispiel (3/4)

- Um die Verfolgungsjagd zu implementieren, wäre es vorteilhaft alle drei **in eine Collection** stecken zu können da wir dann bequem über die einzelnen Charaktere iterieren können.
- Dafür brauchen aber alle Charaktere denselben Typ.
- Leider stecken alle drei Charaktere aber in unterschiedlichen Vererbungshierarchien.

Wozu brauche ich das?
Ein Beispiel (4/4)

- Die Lösung: alle drei implementieren dasselbe Interface **Flyable**
  - `class Parrot implements Flyable { ... }`
  - `class Robot implements Flyable { ... }`
  - `class SuperVillain implements Flyable { ... }`

➔ Alle drei Klassen besitzen nun zusätzlich zu den Typen aus ihrer jeweiligen Vererbungshierarchie auch noch den Typ **Flyable**

➔ Sie können daher alle drei in eine Collection die **Flyable**s aufnimmt eingefügt werden.

  - `LinkedList <Flyable> theChase = new LinkedList <> ();`
  - `Robot ronja =  new Robot();`
  - `Parrot paco = new Parrot(); ...`
  - `theChase.add(paco); ...`
- **Codebeispiel zum Experimentieren in CodeOcean.**

Interfaces

- Jede Klasse kann **beliebig viele** Interfaces implementieren
- Jede **nicht-abstrakte Klasse,** die ein Interface implementiert, muss alle deren Methoden **implementieren**
- Abstrakte Klassen können ebenfalls Interfaces implementieren
  - Die Implementierung der Methoden aus dem Interface kann an die erste nicht-abstrakte Subklasse weitergereicht werden.
- Jede Klasse kann **nur eine Elternklasse** erweitern, daneben aber auch Interfaces implementieren.
  - `class SuperVillain extends Human implements Flyable, Swimmable {}`
- Interfaces können sich **gegenseitig erweitern**.
  - `interface Flyable extends Moveable {}`

Anwendungsbeispiele für Interfaces

- Einfordern von Verhaltensweisen
- Konstantenlisten (Mehr Details im Deep Dive)
- Beispiele aus der Java API:
  - Iterable
  - Comparable
  - ...

Deep Dive Java: Woche 4

openHPI-Java-Team

Hasso-Plattner-Institut

Inhalte Deep Dive Woche 4

- Weitere Modifikatoren: `static` und `final`
- Konstantenlisten
- Wer oder was bin ich? - `instanceof` und `getClass()`
- Short Circuit Evaluation

Statische Methoden und Attribute (1/2)

- Schlüsselwort: `static`, anwendbar auf Attribute und Methoden
- Attribut bzw. Methode ist für die Klasse definiert, nicht für das Objekt
- Für statische Methoden gilt:
  - Es muss kein Objekt von der Klasse instanziiert werden
  - Der Aufruf erfolgt direkt auf der Klasse
  - Beispiele:
    - Math-Klasse: Math.min(…), Math.pow(…), …
    - Wrapper-Klassen: Double.parseDouble(…), …
  - Attribute des Objekts, können nicht gelesen oder geschrieben werden
  - `this.` kann in einem statischen Kontext **nicht** benutzt werden

Statische Methoden und Attribute (2/2)

- Für statische Attribute gilt:
  - Alle Instanzen der Klasse (Objekte) greifen auf denselben Wert zu
  - Nicht-statische Methoden können statische Attribute lesen und schreiben
  - Beispiel:

```
1 public class Robot {
2    public static int roboCount;
3    public Robot() {
4       roboCount++;
5    }
6    public int noOfRobots() {
7       return roboCount;
8    }
9 }
```

- Jeder neue Roboter erhöht den Zähler um eins (für alle Roboter)

Finale Klassen, Methoden und Konstanten

- Schlüsselwort `final`, anwendbar für Klassen, Attribute und Methoden
- Finale **Methoden** können nicht überschrieben werden
- Finale **Klassen** können nicht erweitert werden
- Finale **Attribute** können nach der Initialisierung nicht mehr neu beschrieben werden ➔ **Konstanten**
- Beispiel:
  - Integer.*MAX_VALUE*
  - `public final String PRODUCER = "Daniels Roboterfabrik";`
- Konventionen:
  - Konstanten schreibt man in Großbuchstaben
  - Underscore ersetzt CamelCase

Interfaces und Konstanten

- Mögliche Elemente in Interfaces
  - Methoden: `public abstract`
  - Attribute: `public static final`
- Interfaces werden daher manchmal benutzt, um Programmweite Konstanten zu definieren

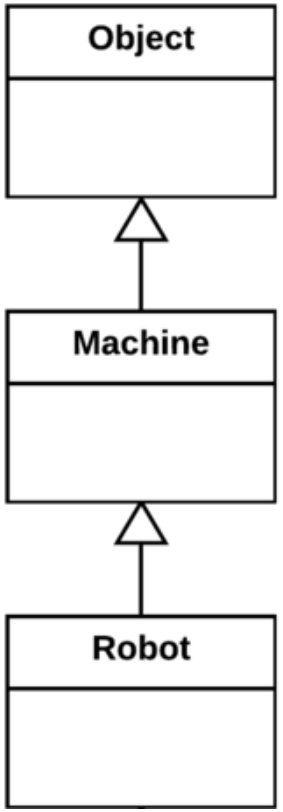
```
1 public interface Config {
2    public static int MEANING_OF_LIFE = 42;
3 }
```

- Dies gilt aus verschiedenen Gründen als sogenanntes Anti-Pattern (schlechter Stil). Bessere Lösung:

```
1 public final class Config {
2    private Config() {}
3    public static int MEANING_OF_LIFE = 42;
4 }
```

Welchen Typ hat mein Objekt?
Aus welcher Klasse wurde mein Objekt instanziiert

- Manchmal will ich wissen welchen Typ mein Objekt hat bzw. aus welcher Klasse es instanziiert wurde
- Java bietet hierfür 2 Optionen
  - `instanceof` Operator
  - `getClass()` Methode



- Unterschiedliches Verhalten

```
1 Machine robot = new Robot("Ronja");
2 robot.getClass() ➔  Robot
3 robot instanceof Robot ➔ true
4 robot instanceof Machine ➔ true
5 robot instanceof Object ➔ true
6
7 Machine machine = new Machine ("Ronja");
8 machine instanceof Robot ➔ false
```

Welchen Typ hat mein Objekt?
Aus welcher Klasse wurde mein Objekt instanziiert

- Problem: `equals(...)` Methode

- Zur Erinnerung:

```
1 @Override
2 public boolean equals(Object o) {
3     // [...] sicherstellen dass o vom richtigen Typ ist
4     return (o != null) &&
5            this.name.equals(((Parrot) o).name) &&
6            this.age == ((Parrot) o).age;
7 }
```

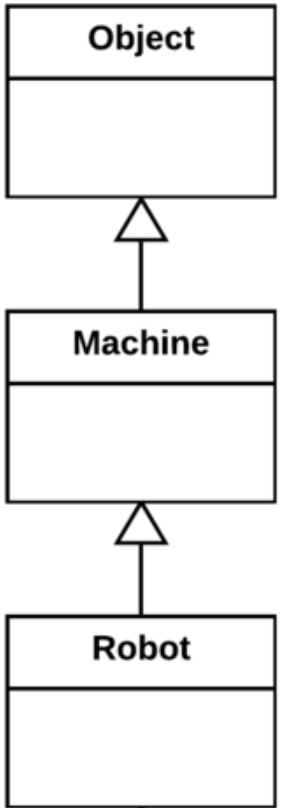
- **Jetzt richtig:**

```
1  @Override
2  public boolean equals(Object obj) {
3      if (this == obj) { return true; }
4      if (obj == null) { return false; }
5
6      // sicherstellen dass o vom richtigen Typ ist
7      // entweder
8      if (!(obj instanceof Robot)) { return false; }
9      // oder
10     if (getClass() != obj.getClass()){ return false; }
11
12     Robot other = (Robot) obj;
13     if (name == null) {
14             if (other.name != null) { return false; }
15     } else if (!name.equals(other.name)) { return false; }
16     return true;
   }
```

Welchen Typ hat mein Objekt?
Aus welcher Klasse wurde mein Objekt instanziiert

- Problem: Beide Lösungen sind nicht optimal

```
1  Machine machine = new Machine("Ronja");
2  Machine robot = new Robot("Ronja");
3
4  if (!(obj instanceof Robot)) { return false; }
5
6  machine.equals(robot) ➔ true
7  robot.equals(machine) ➔ false
8
9
10 if (getClass() != obj.getClass()) { return false; }
11
12 machine.equals(robot) ➔ false
13 robot.equals(machine) ➔ false
```



&& Short Circuit Evaluation

- Wiederholung Boolsche Ausdrücke (Wahr oder Falsch):
  - UND (&&) gesamter Ausdruck ist falsch, wenn ein Teil falsch ist
  - ODER (||) gesamter Ausdruck ist wahr, wenn ein Teil wahr ist
  - Java nutzt das aus, um Bedingungen abzubrechen die nicht mehr **wahr**, bzw. **falsch** werden können
- Short Circuit Evaluation (deutsch: Kurzschlussauswertung)
  - &&-verknüpften Ausdrücke werden bei erstem falschen Teilausdruck abgebrochen
  - ||-verknüpfte Ausdrücke werden bei erstem richtigen Teilausdruck abgebrochen
  - Geschickte Anordnung der Teilausdrücke kann die Abfragen vereinfachen

```
return (o != null) &&
        this.name.equals(((Parrot) o).name);
```

Programming Toolbox

Hasso Plattner Institut

final

Collections

Wrapper

Type Casting

Objektdatentypen

static

foreach

Unser erstes und letztes Programmierbeispiel

openHPI-Java-Team

Hasso-Plattner-Institut

Unser erstes Programmierbeispiel

```
1  class HelloPaco {
2     public static void main(String[] args){
3        System.out.println("Hallo Paco");
4     }
5  }
```

Ausgabe:

```
Hallo Paco
```


```
1  class HelloPaco {
2     public static void main(String[] args){
3        System.out.println("Hallo Paco");
4     }
5  }
```

Klassen

- Schlüsselwort `class`
- Baupläne für (mehr oder weniger abstrakte) Konstrukte aus der realen Welt
- Verfügen über Zustand (Attribute) und Verhalten (Methoden)
- Klassenbezeichner kann frei gewählt werden und wird groß (CamelCase) geschrieben
- Verwandte Konzepte: Vererbung, Abstrakte Klassen, Interfaces


```
1  class HelloPaco {
2     public static void main(String[] args){
3        System.out.println("Hallo Paco");
4     }
5  }
```

Sichtbarkeiten

- Regeln Zugriff auf Elemente
- "Sichtbarkeit" - bezieht sich auf den Kontext (wo ist was sichtbar)
- `public` = uneingeschränkter Zugriff von überall
- main Methode muss immer `public` sein
- Alternativen für alle anderen Methoden: `protected`, `private`, default
- Attribute werden in der Regel als `private` deklariert
- Regelung der Lese- und Schreibrechte über Getter/Setter


```
1  class HelloPaco{
2     public static void main(String[] args){
3        System.out.println("Hallo Paco");
4     }
5  }
```

Statische Methoden und Attribute

- Element der Klasse nicht des Objekts
  - Keine Objekt-Instantiierung mit `new`
  - Aufruf direkt auf der Klasse: `Double.parseDouble("22.8")`
  - Attribute: gemeinsamer Wert für alle Objekte der Klasse


```
1  class HelloPaco{
2     public static void main(String[] args){
3        System.out.println("Hallo Paco");
4     }
5  }
```

Rückgabetypen:

- Datentyp des Werts mit dem die Methode antwortet
- `void` = kein Rückgabewert
- `main` immer `void`
- Konstruktoren geben immer ein Objekt ihrer Klasse zurück
- Andere Methoden können `void` oder beliebige primitive oder Objekt-Datentypen zurückgeben


```
1  class HelloPaco{
2     public static void main(String[] args){
3        System.out.println("Hallo Paco");
4     }
5  }
```

Methodenbezeichner:

- Individueller Name für Methode
- `main` ist reserviert und darf nur einmal pro Programm vorkommen
- `main`-Methode startet Programmfluss
- Bezeichner für andere Methoden können nahezu frei gewählt werden
  - Darf nicht mit Zahl beginnen, keine Sonderzeichen
  - Kleingeschrieben (camelCase)
  - Ausnahme: Konstruktor (wie Klassenname → CamelCase)


```
1  class HelloPaco{
2     public static void main(String[] args){
3        System.out.println("Hallo Paco");
4     }
5  }
```

Parameter:

- Variablen für Argumente, die von der Methode beim Aufruf erwartet werden
- Mittels des Bezeichners kann innerhalb der Methode auf den beim Methodenaufruf übergebenen Wert zugegriffen werden


```
1  class HelloPaco{
2     public static void main(String[] args){
3        System.out.println("Hallo Paco");
4     }
5  }
```

Array:

- Container fester Größe für Objekte (oder primitive) gleichen Datentyps
- Gekennzeichnet durch <Datentyp>[]
- `String[] args` = User-Eingabe beim Programmstart
  - z.B. Kommandozeilenargumente


```
1  class HelloPaco{
2     public static void main(String[] args){
3        System.out.println("Hallo Paco");
4     }
5  }
```

Ausgaben auf der Konsole

- `System.out.println(...)`
  - Überladen für alle primitiven Datentypen und String
  - Objektdatentypen werden mittels toString() als String ausgegeben
  - Gibt einzelne Zeile aus (mit Zeilenumbruch am Ende)
- `System.out.print(...)`
  - Gibt einzelne Zeichenkette aus (ohne Zeilenumbruch am Ende)


```
1  class HelloPaco{
2     public static void main(String[] args){
3        System.out.println("Hallo Paco");
4     }
5  }
```

Argumente:

- Übergeben einen Wert an den Parameter der aufgerufenen Methode
- Als Argument übergeben werden können:
  - Werte: `add(3, 4);` `println("Hallo Paco");`
  - Variablen: `add(x, y);`
  - Methodenaufrufe: `println(getName());`
    `println(paco.getName());`


```
1  class HelloPaco{
2     public static void main(String[] args){
3        System.out.println("Hallo Paco");
4     }
5  }
```

Anweisung

- Befehl an den Computer etwas auszuführen


```
1  class HelloPaco{
2     public static void main(String[] args){
3        System.out.println("Hallo Paco");
4     }
5  }
```

Semikolon

- Beendet Anweisung
- Häufige Fehlerquelle: Semikolon vergessen


```
1  class HelloPaco {
2     public static void main(String[] args){
3        System.out.println("Hallo Paco");
4     }
5  }
```

CodeBlocks

- Begrenzt durch { }
- Definieren Scopes (Gültigkeitsbereiche)
  - Klassen
  - Methoden
  - Schleifen / Bedingungen
  - Einfach nur Blocks (selten genutzt)