



Objektorientierte Programmierung mit Java – Woche 3

openHPI-Java-Team
Hasso-Plattner-Institut



Wiederholung Woche 2

openHPI-Java-Team
Hasso-Plattner-Institut



```
1 double add(int a, int b, double c){  
2     return a+b+c;  
3 }
```

Aufruf: `double result = add(3,4,1.1);`

Syntax:

<Rückgabetyp><Methodenbezeichner>(<Datentyp><Parameterbezeichner>, ...)

- **Parameter:** Bei der Methodendefinition
- **Argument:** Übergabewert beim Aufruf der Methode



```
1  if (robin.isBatteryLow() && robin.isInDockingStation())
2  {
3      robin.chargeBattery();
4  } else if (robin.isBatteryLow()) {
5      robin.driveToDockingStation();
6  } else {
7      robin.doThings();
8  }
```

- Ausführungsreihenfolge von oben nach unten
- Datentyp: **boolean** (kann entweder **true** oder **false** sein)
- Verwendet um Bedingungen zu überprüfen



```
1 for (int i = 0; i < 3; i++) {  
2     //...  
3 }
```

Schleifenbedingung
Schleifenkopf
Schleifenrumpf

```
1 while ( i < j ){  
2     //...  
3 }
```

Schleifenbedingung
Schleifenkopf
Schleifenrumpf



Arrays

```
1  int[] numbers = new int[5];  
2  //int[] numbers = {0,0,0,0,0};  
3  numbers[0] = 6;  
4  numbers[1] = 33;  
5  numbers[2] = 9;  
6  numbers[3] = 0;  
7  numbers[4] = 503;
```

- **Achtung!** In Java fängt man mit der 0 an zu zählen



```
1  class Robot{  
2      String name;  
3  
4      Robot(String name){  
5          this.name = name;  
6      }  
7  }
```

- Gleicher Bezeichner wie Klasse
- **this** wird verwendet, um auf Attribute dieses Objekt zuzugreifen
- Aufruf: Robot robin = new Robot("Robin");



Vererbung

openHPI-Java-Team
Hasso-Plattner-Institut

Warum Vererbung?

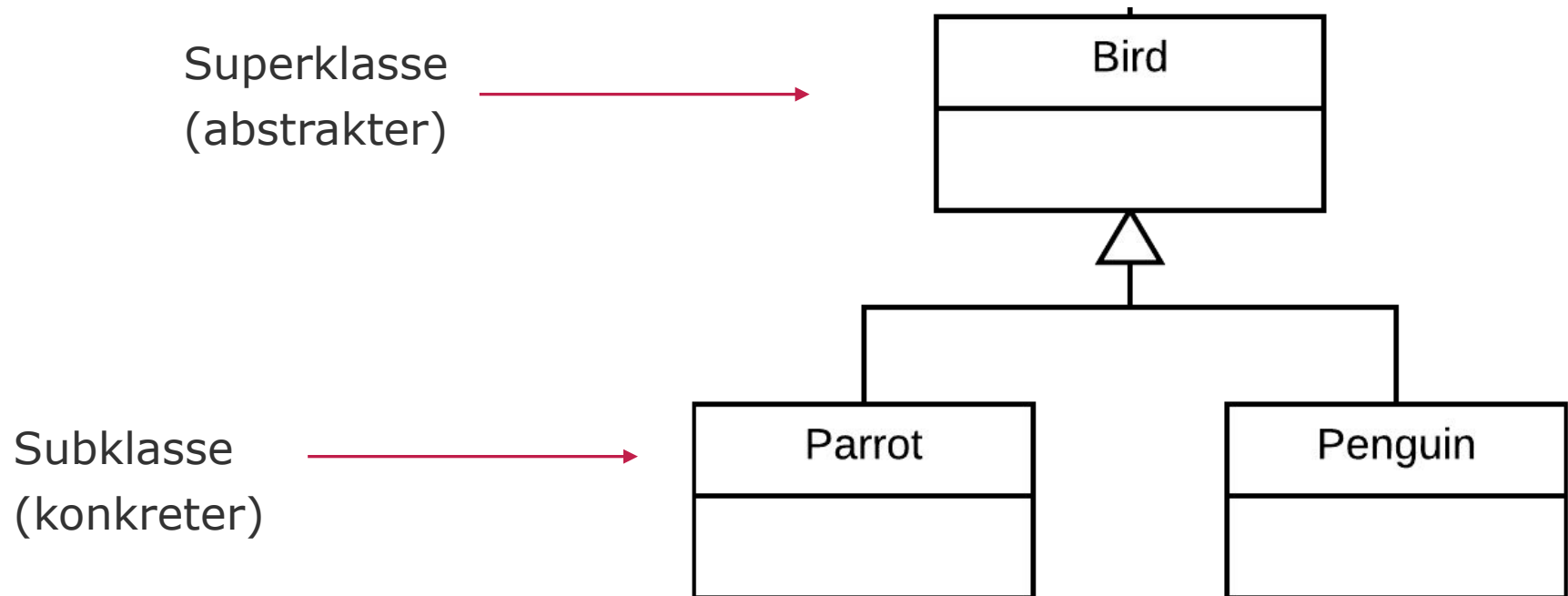


Parrot
name:String species:String dateofBirth: Date featherColors: String
fly() eat() sleep() talk()

Penguin
name:String species:String dateofBirth: Date
swim() eat() sleep()



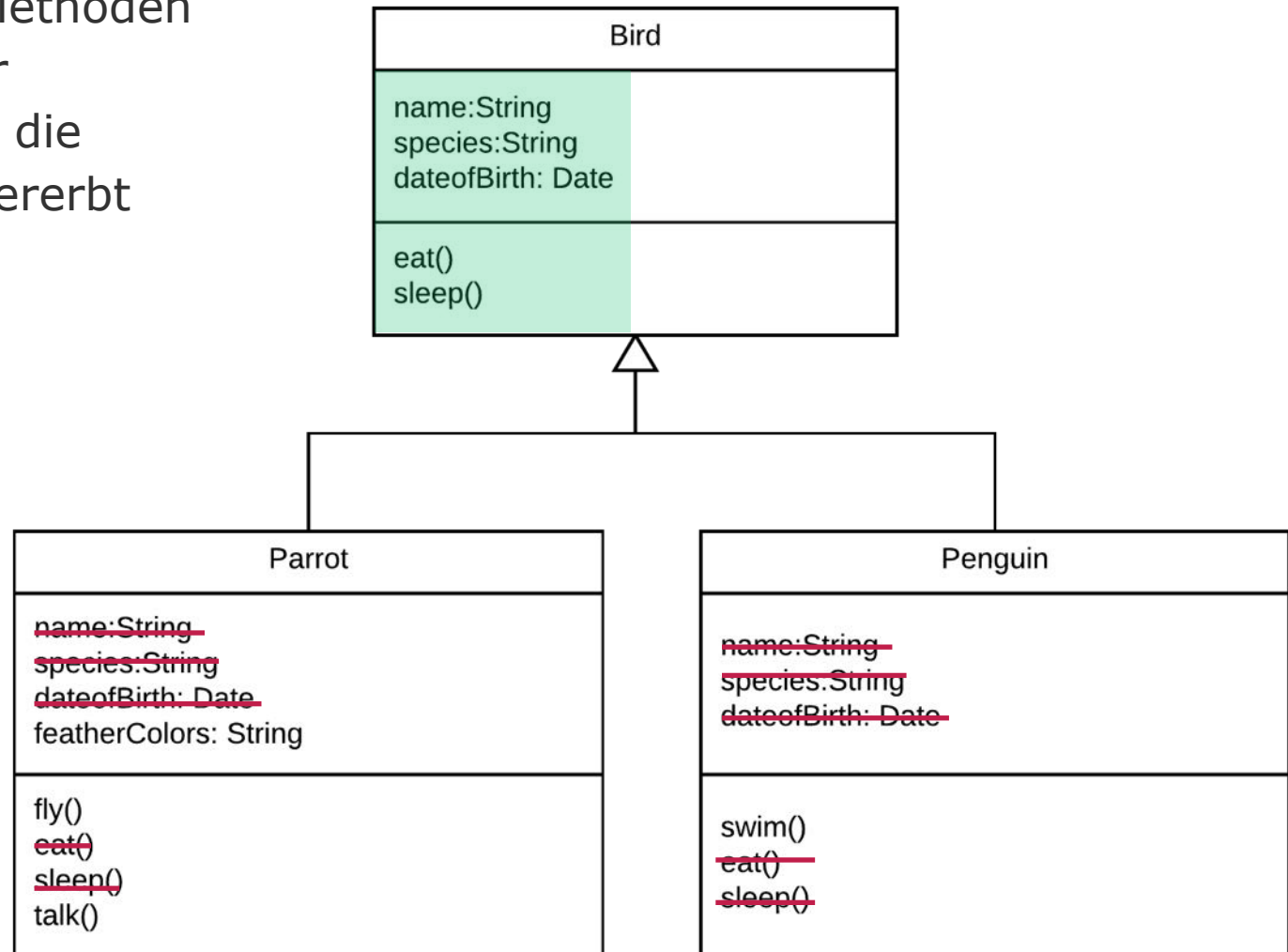
- Ein Papagei ist ein Vogel
- Ein Pinguin ist ein Vogel



Superklasse Bird

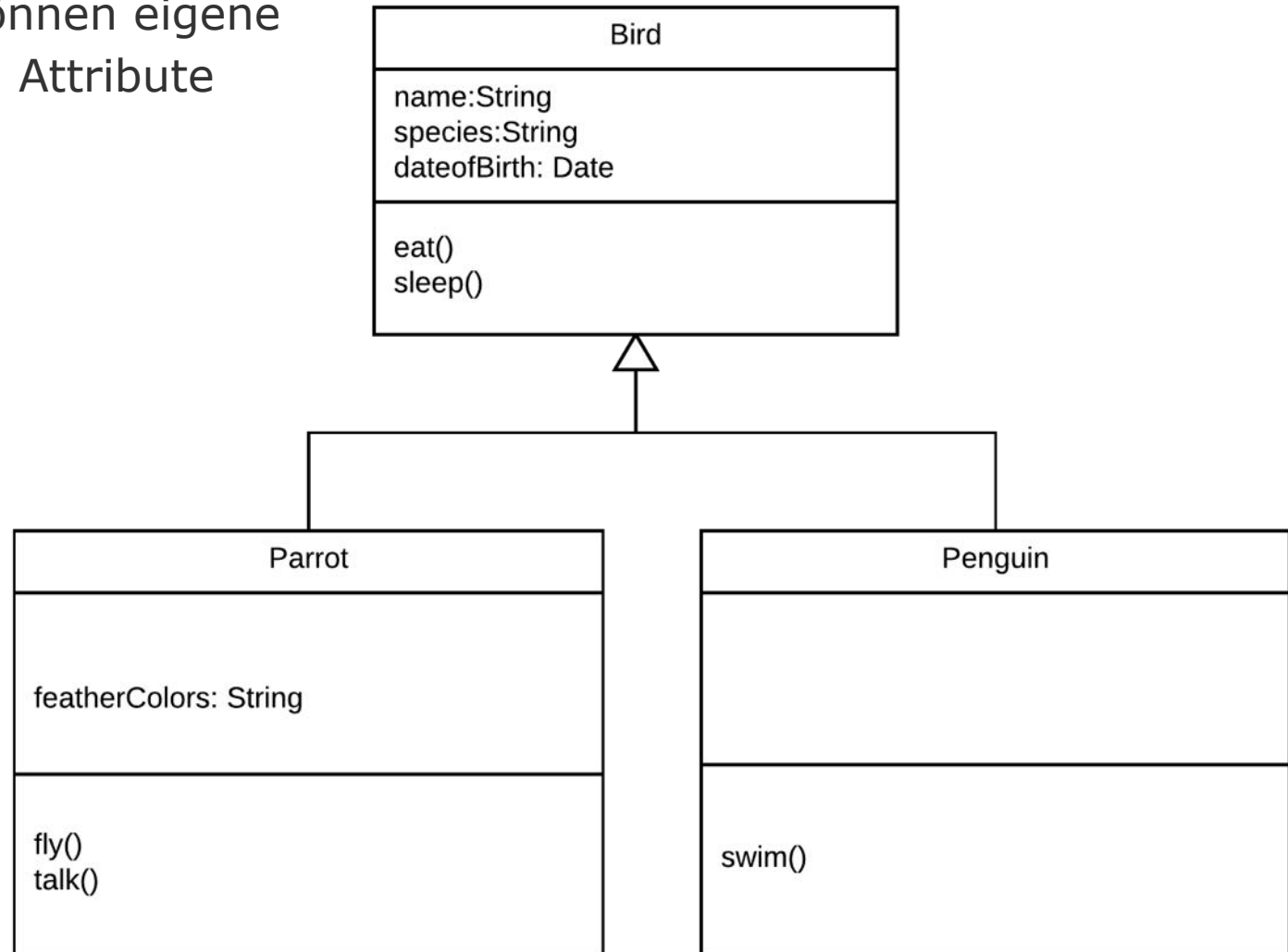


- Attribute und Methoden werden von der Superklasse an die Subklasse(n) vererbt





- Subklassen können eigene Methoden und Attribute ergänzen





Schlüsselwort: **extends**

```
1 class Bird {  
2     String name;  
3  
4     void sayHello() {  
5     }  
6 }  
1 class Parrot extends Bird {  
2 }
```

Subklassen

- Werden mit **extends** angelegt
- Syntax: **class** <subclass> **extends** <superclass> { }



```
1 class Bird {  
2     String name;  
3  
4     void sayHello() {  
5     }  
6 }  
1 class Parrot extends Bird {  
2 }
```

Subklasse

- Enthält implizit alle Methoden (sayHello()) der Superklasse
- Enthält implizit alle Attribute (name) der Superklasse



Verwendung der Subklasse

```
1 class Story {
2     public static void main(String[] args) {
3         Parrot paco = new Parrot();
4         paco.name = "Paco";
5     }
6 }

1 class Bird {
2     String name;
3     // ...
4 }

1 class Parrot extends Bird {
2 }
```

- Die Klasse Parrot hat implizit das Attribut `name` der Klasse Bird



Erweiterung der Subklasse

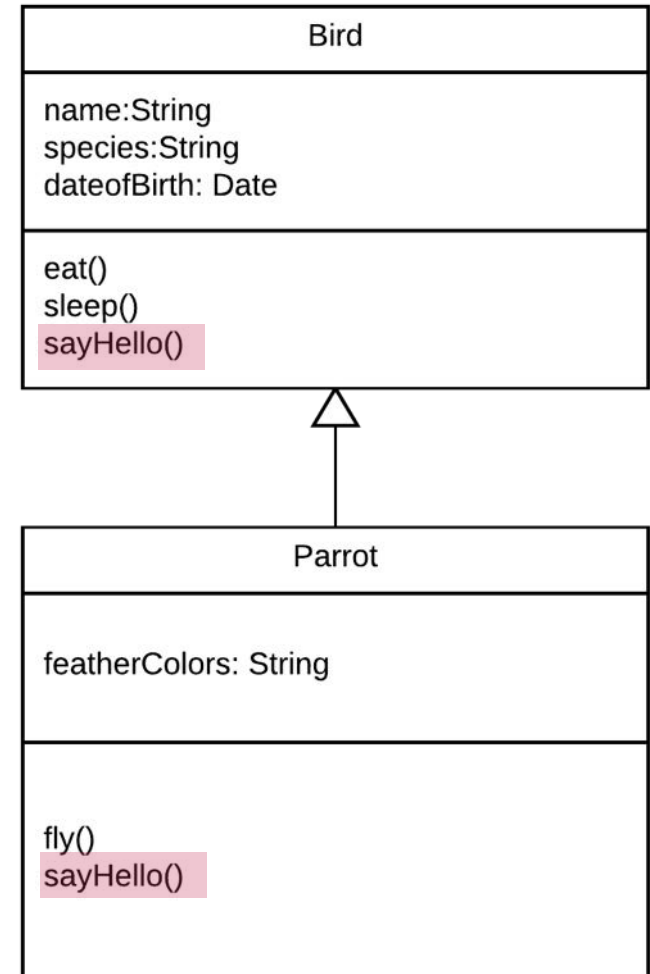
```
1 class Bird {  
2     String name;  
3  
4     void sayHello() {  
5     }  
6 }  
1 class Parrot extends Bird {  
2     String featherColor;  
3 }
```

Subklasse

- Kann Attribute ergänzen (`featherColor`)
- Kann Methoden ergänzen



- Subklassen können Methoden überschreiben und damit geerbtes Verhalten individuell verändern





```
1 class Bird {  
2     String name;  
3  
4     void sayHello() {  
5     }  
6 }
```

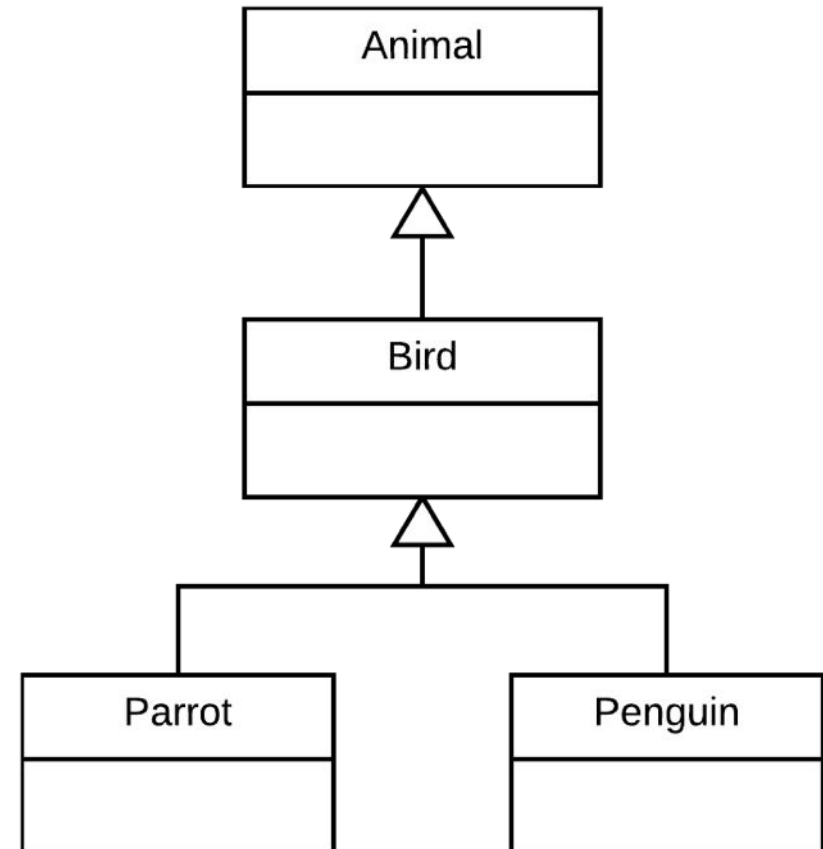
```
1 class Parrot extends Bird {  
2     @Override  
3     void sayHello() {  
4         System.out.println("Hallo Ann Katrin und Tom!");  
5     }  
6 }
```

Subklasse

- Kann Attribute überschreiben
- Kann Methoden überschreiben (sayHello())
- Auch Override genannt



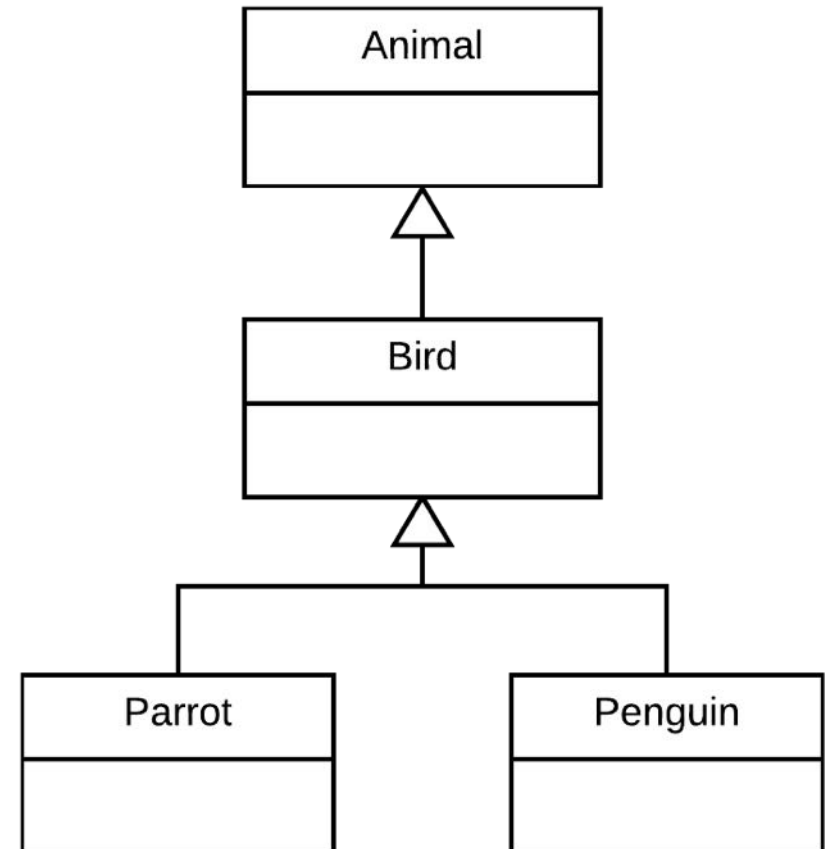
- Ein Papagei **ist ein** Vogel
- Ein Pinguin **ist ein** Vogel
- Ein Vogel **ist ein** Tier
- *Is-a Beziehung* in einem UML Klassendiagramm
 - A parrot **is a** bird
 - A parrot **is an** animal
- Die Klasse Parrot erhält transitiv alle Methoden und Attribute der Klasse Animal



In welche Richtung wird vererbt?



Achtung! Vererbung funktioniert nur in eine Richtung. Ein Papagei ist ein Tier. Aber ein Tier ist **kein** Papagei.





Sichtbarkeiten

openHPI-Java-Team

Hasso-Plattner-Institut



```
1 public class HelloPaco{  
2     public static void main(String[] args){  
3         // ...  
4     }  
5 }
```

Sichtbarkeiten

- Regeln Zugriff auf Elemente
- Sichtbar: Element kann gelesen/geschrieben werden
- Nicht sichtbar: Element kann nicht gelesen und verändert werden
- Innerhalb einer Klasse sind immer alle Elemente der Klasse sichtbar



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



Beispiel `public`

```
1 public class Parrot{  
2     public String name;  
3 }
```

```
1 public class Story{  
2     public static void main(String[] args) {  
3         Parrot paco = new Parrot();  
4         paco.name = "Paco";  
5     }  
6 }
```

Public

- Uneingeschränkte Sichtbarkeit
- Lesen und schreiben überall möglich
- Einsatz: für öffentliche Methoden, nicht für Attribute



Beispiel `protected`

```
1 public class Parrot{
2     protected String favoriteFood;
3 }

1 public class Story{
2     public static void main(String[] args) {
3         Parrot paco = new Parrot();
4         paco.favoriteFood = "Äpfel";
5     }
6 }
```

Protected

- Zugriff nur aus der eigenen Klasse, der Subklasse oder dem Package



Beispiel `protected`

```
1 public class Parrot{
2     protected String favoriteFood = "Äpfel";
3
4     public void eat() {
5         System.out.println("Paco isst gerade " +
6             this.favoriteFood);
7     }
8 }
```

Protected

- Zugriff nur aus der eigenen Klasse, der Subklasse oder dem Package



Beispiel `protected`

```
1 public class Bird{
2     protected String name;
3     protected String favoriteFood;
4 }
```

```
1 public class Parrot extends Bird {
2     void eat() {
3         System.out.println(this.name + " isst " +
4             this.favoriteFood);
5     }
6 }
```

Protected

- Zugriff nur aus der eigenen Klasse, der Subklasse oder dem Package



Beispiel `private`

```
1 public class Parrot {  
2     private String name;  
3 }
```

```
1 public class Story{  
2     public static void main(String[] args) {  
3         Parrot paco = new Parrot();  
4         paco.name = "Paco";  
5     }  
6 }
```

Private

- Stark eingeschränkte Sichtbarkeit
- Kein Zugriff von außen
- Nur Zugriff innerhalb der Klasse



Beispiel `private`

```
1 public class Parrot {  
2     private String name;  
3  
4     Parrot(String name) {  
5         this.name = name;  
6     }  
7  
8     public void greet(Parrot other) {  
9         System.out.println("Hallo " + other.name);  
10    }  
11 }
```

Private

- Bei zwei Objekten einer Klasse ist gegenseitiger Zugriff möglich



Beispiel `private`

```
1 public class Story{
2     public static void main(String[] args) {
3         Parrot paco = new Parrot("Paco");
4         Parrot polly = new Parrot("Polly");
5         polly.greet(paco);
6     }
7 }
```

Private

- Bei zwei Objekten einer Klasse ist gegenseitiger Zugriff möglich

Ausgabe:

Hallo Paco



	Innerhalb der Klasse	Im Package	Subklassen	Sonstige Klassen
private	Ja	Nein	Nein	Nein
(default)	Ja	Ja	Nein	Nein
protected	Ja	Ja	Ja	Nein
public	Ja	Ja	Ja	Ja



Public	Attribute und Methoden werden weitervererbt
Protected	Attribute und Methoden werden weitervererbt
default (Package Private)	Attribute und Methoden werden nur innerhalb eines Packages weitervererbt
Private	Attribute und Methoden werden nicht weitervererbt



Kapselung

openHPI-Java-Team
Hasso-Plattner-Institut



```
1 public class Parrot {  
2     String name;  
3     int age;  
4     boolean alive;  
5 }
```

Kapselung

- Verhindert, dass direkter Zugriff von außen erfolgt
- Schützt Attribute vor unkontrollierter Veränderung



```
1 class Detective {  
2     private String pinCode;  
3 }  
4
```

Kapselung

- Verhindert, dass ungewollter Zugriff auf Werte erfolgt



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 }
```

Private

- 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



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 }
```

Getter und Setter

- Das Attribut kann von außen gelesen und geschrieben werden



```
1 public class Parrot {  
2     private String name;  
3  
4     public String getName() {  
5         return this.name;  
6     }  
7 }
```

Getter

- Das Attribut kann von außen gelesen werden



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

Setter

- Das Attribut kann von außen geschrieben werden



Validierung von Werten

```
1 public class Parrot {  
2     protected int age;  
3  
4     public void setAge(int newAge) {  
5         if (newAge > 0) {  
6             this.age = newAge;  
7         } else {  
8             this.age = 1;  
9         }  
10    }  
11 }
```

Kapselung

- Ermöglicht die Validierung von Attribut-Werten



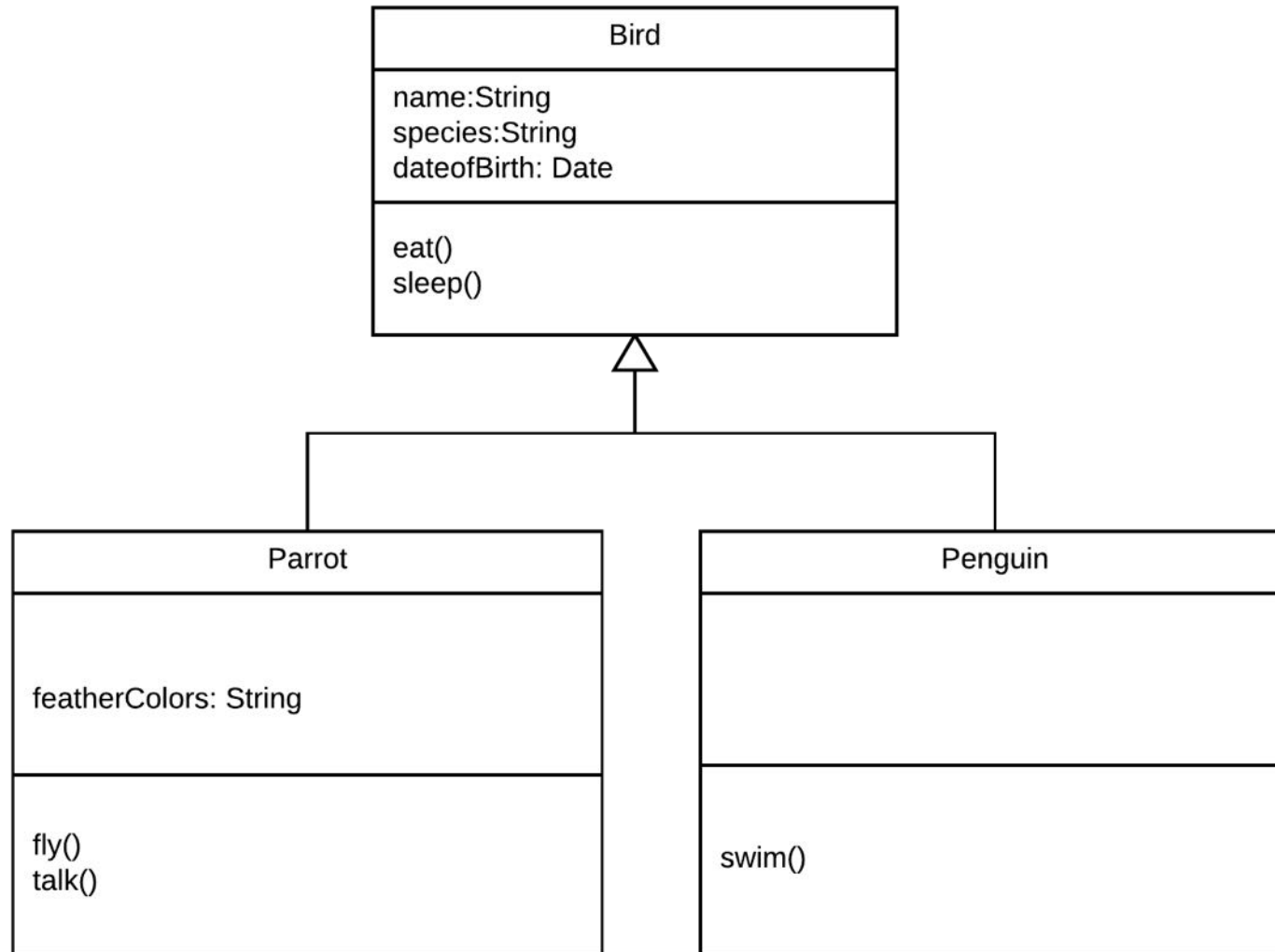
```
1 public class Robot {  
2     private boolean stealthMode = false;  
3  
4     public void enableStealthMode() {  
5         stealthMode = true;  
6         this.log("Tarnmodus aktiviert");  
7     }  
8  
9     private void log(String logMessage) {  
10        sendDataToDuke(logMessage);  
11    }  
12 }
```

- Die Implementierung kann geändert werden ohne die Zusammenarbeit mit anderen Klassen zu beeinträchtigen
- Kapselung sorgt für Modularisierung

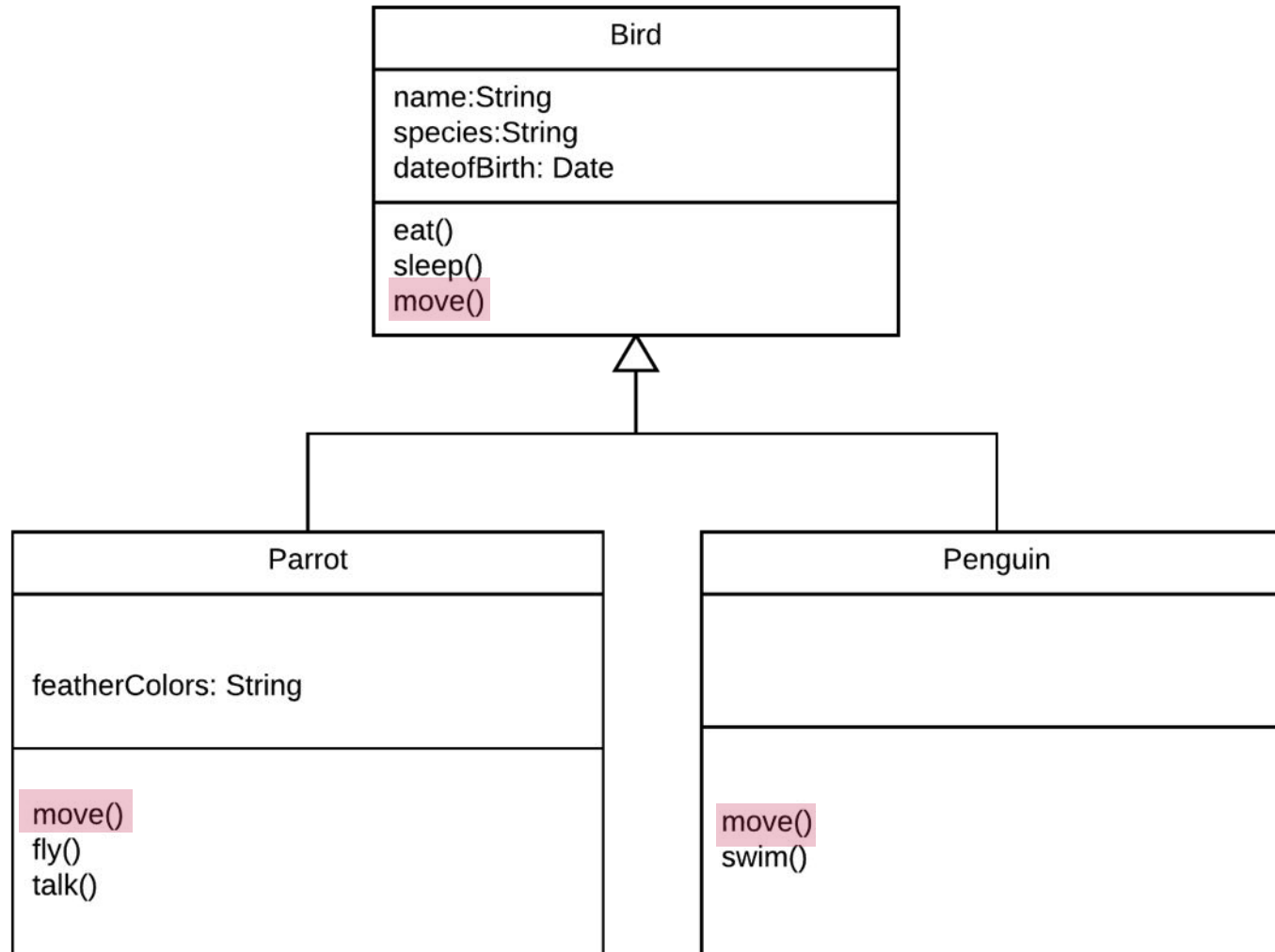


Überschreiben von Methoden (Override)

openHPI-Java-Team
Hasso-Plattner-Institut



Überschreiben der Methode move()





```
1 public class Bird {  
2  
3     public void move() {  
4         System.out.println("Ich bewege mich");  
5     }  
6 }
```

```
1 public class Parrot extends Bird {  
2     @Override  
3     public void move() {  
4  
5         this.fly();  
6     }  
7     private void fly() {  
8         System.out.println("Ich fliege");  
9     }  
10 }
```



```
1 public class Bird {  
2  
3     public void move() {  
4         System.out.println("Ich bewege mich");  
5     }  
6 }
```

```
1 public class Parrot extends Bird {  
2     @Override  
3     public void move() {  
4         super.move();  
5         this.fly();  
6     }  
7     private void fly() {  
8         System.out.println("Ich fliege");  
9     }  
10 }
```



Aufruf von Methoden der Superklasse

```
1 public class Parrot extends Bird {  
2     @Override  
3     public void move() {  
4         super.move();  
5         this.fly();  
6     }  
7  
8     private void fly() {  
9         System.out.println("Ich fliege");  
10    }  
11 }
```

- `super.<methodenBezeichner>()`; ermöglicht den Aufruf von Methoden der Superklasse
- `this.<methodenBezeichner>()`; ermöglicht den Zugriff auf Methoden der eigenen Klasse
- Konvention: `@Override` vor die Methode schreiben um zu signalisieren, dass man die Methode überschreibt



Superklasse:

```
public void move() {  
    //...  
}
```

Subklasse:

```
@Override  
public void move() {  
    //...  
}
```

Override

- Darf die Sichtbarkeit nicht stärker einschränken
- Methodenparameter müssen exakt gleich bleiben
- Rückgabetypen müssen kompatibel sein



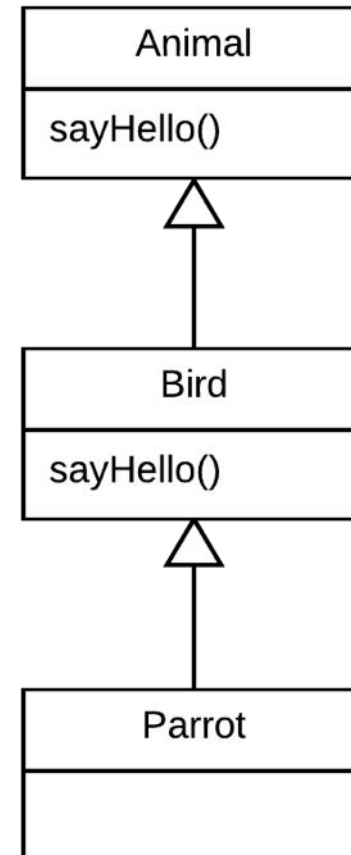
Welche Methode wird genutzt?

Auf einem Objekt der Klasse Parrot wird die Methode sayHello() aufgerufen.

Welche Methode wird aufgerufen?

```
Parrot paco = new Parrot();  
paco.sayHello();
```

Antwort: Immer die speziellste.
In diesem Fall also die Methode sayHello() der Klasse Bird.





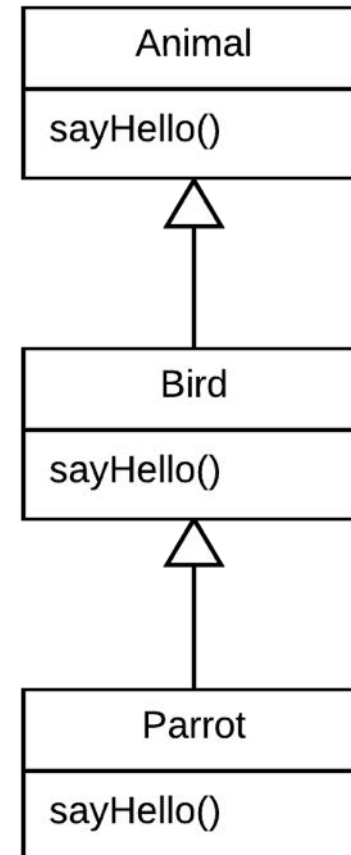
Welche Methode wird genutzt?

Auf einem Objekt der Klasse Parrot wird die Methode sayHello() aufgerufen.

Welche Methode wird aufgerufen?

```
Parrot paco = new Parrot();  
paco.sayHello();
```

Antwort: Die Methode der Klasse Parrot, da diese am speziellsten ist.





Überladung von Methoden (Overload)

openHPI-Java-Team
Hasso-Plattner-Institut



```
1  int sum(int x, int y) {  
2      return x + y;  
3  }  
4  
5  int sum(int x, int y, int z) {  
6      return x + y + z;  
7  }  
8  
9  double sum(double x, double y) {  
10     return x + y;  
11 }
```

Überladung von Methoden

- Gleicher Name
- Unterschiedliche Parameterliste
 - Anzahl der Parameter
 - Datentypen der Parameter



```
1  int sum(int x, int y) {  
2      return x + y;  
3  }  
4  
5  int sum(int c, int d) {  
6      return c + d;  
7  }
```

Überladung von Methoden

- Keine valide Überladung
- Parameterbezeichner sind nicht entscheidend



```
1  int sum(int x, int y) {  
2      return x + y;  
3  }  
4  
5  double sum(int x, int y) {  
6      return x + y;  
7  }
```

Überladung von Methoden

- Keine valide Überladung
- Rückgabetypen nicht entscheidend



```
1  int sum(int x, int y) {  
2      return x + y;  
3  }  
4  
5  int sum(int x, int y, int z) {  
6      return x + y + z;  
7  }  
8  
9  double sum(double x, double y) {  
10     return x + y;  
11 }
```

Was wird aufgerufen?

sum(2, 4)	→ int sum(int x, int y)
sum(2.0, 4.0)	→ double sum(double x, double y)
sum(2.3, 4)	→ double sum(double x, double y)
sum(2.0, 4)	→ double sum(double x, double y)



```
1  int sum(int x, int y) {  
2      return x + y;  
3  }  
4  
5  int sum(int x, int y, int z) {  
6      return x + y + z;  
7  }  
8  
9  double sum(double x, double y) {  
10     return x + y;  
11 }
```

Überladung von Methoden

- Es wird die am Besten übereinstimmende Methode ausgeführt

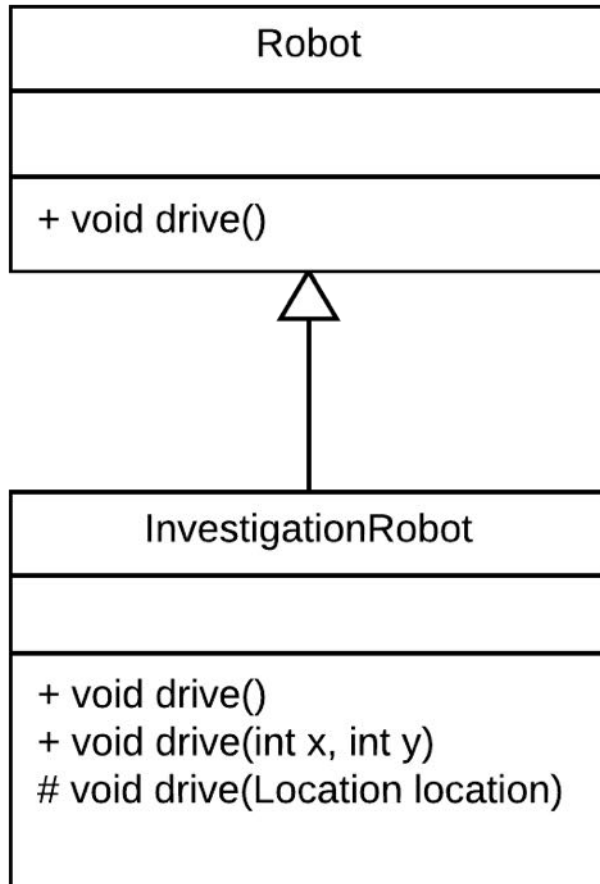


Ein Beispiel aus der Java API

```
1 public class PrintStream{
2     void print(int arg) { ...}
3     void print(String arg) { ...}
4     void print(char[] arg) { ...}
5     // ...
6 }
```

Vorteile:

- Methodenbezeichner wird identisch wiederverwendet
 - Nicht printInt(), printString() etc.
- Ausführung variiert abhängig von den übergebenen Argumenten



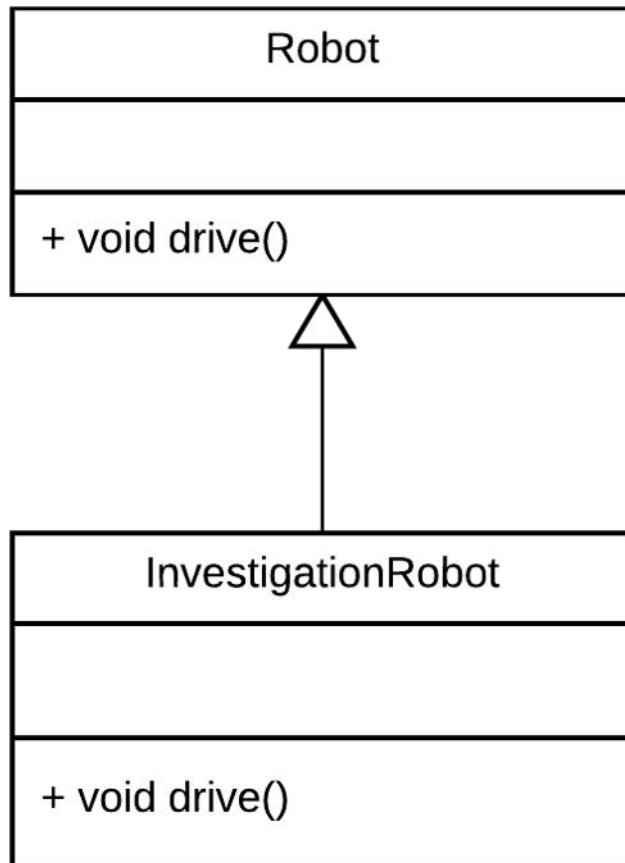
Überladen

- Verschiedene Methoden haben den gleichen Namen innerhalb einer Klasse
- Hat **nichts** mit Vererbung zu tun
- Parameterliste muss verändert werden
- Rückgabetypen können unterschiedlich sein
 - Ändern des Rückgabtyps reicht alleine nicht aus
- Sichtbarkeit darf in jede Richtung verändert werden



Überschreiben

- Methode wird von der Superklasse an die Subklasse vererbt
- Subklasse ändert ihr Verhalten in Bezug auf diese Methode
- Parameter müssen gleich bleiben
- Rückgabetypen müssen kompatibel sein
- Sichtbarkeit darf nicht weiter eingeschränkt werden





Abstrakte Klassen

openHPI-Java-Team
Hasso-Plattner-Institut

Motivation abstrakte Klassen



■ Parrot paco = new Parrot();



■ Robot ronja = new Robot();



■ Animal animal = new Animal();



■ Manche Klassen sollten nicht instanziiert werden können!



```
1 abstract public class Animal {  
2     String name;  
3  
4     public void move() {  
5     }  
6 }
```

Abstrakte Klassen

- Werden durch das Schlüsselwort **abstract** angelegt
- Können nicht instanziiert werden
- Müssen "gesubclasst" werden um von diesen Subklassen Instanzen erzeugen zu können



```
1 public abstract class Animal {  
2     String name;  
3  
4     public void move() {  
5     }  
6 }
```

Abstrakte Klassen

- Reihenfolge ob `public abstract` oder `abstract public` ist nicht entscheidend

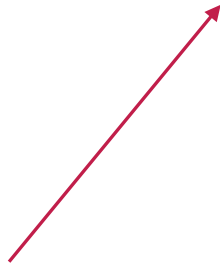


```
1 abstract public class Bird extends Animal {  
2  
3     public void fly() {  
4     }  
5 }
```

- Auch Subklassen von abstrakten Klassen können abstrakt sein



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



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)
- Abstrakte Methoden können nur in abstrakten Klassen existieren



```
1 public class Parrot extends Bird {  
2     String name;  
3  
4     @Override  
5     public void eat() {  
6         System.out.println("Ich esse");  
7     }  
8 }
```

Abstrakte Methoden

- Methode `eat()` muss von der ersten konkreten Subklasse überschrieben werden



Polymorphie

openHPI-Java-Team

Hasso-Plattner-Institut



Eine Papageienliste

```
[...]  
1  Parrot paco  = new Parrot();  
2  Parrot polly = new Parrot();  
3  Parrot penny = new Parrot();  
4  
5  Parrot[] array = new Parrot[3];  
6  array[0] = paco;  
7  array[1] = polly;  
8  array[2] = penny;  
9  
10 for (int i = 0; i < array.length; i++) {  
11     array[i].move();  
12 }  
[...]
```

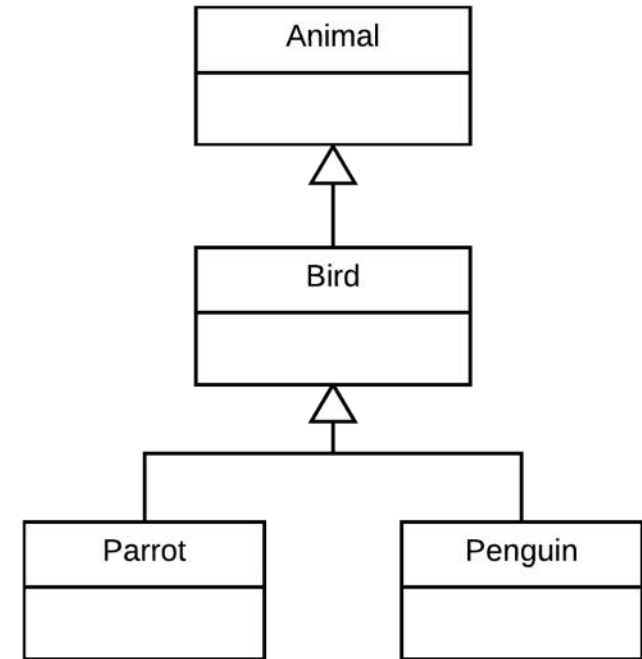
Ausgabe:

```
Ich fliege  
Ich fliege  
Ich fliege
```

Auch Papageien sind Vögel



```
[...]  
1   Bird penguin = new Penguin();  
2   Bird parrot = new Parrot();  
[...]
```



- Einer Variable vom Typ Bird können Objekte aller Unterklassen von Bird zugewiesen werden
- Ein Objekt der Klasse Bird kann auch nur Methoden der Klasse Bird aufrufen



```
[...]  
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  }  
[...]
```

Wir können zwar keine Objekte vom Typ `Bird` instanziiieren (da dies eine abstrakte Klasse ist), aber wir können ein Array erzeugen, dass Objekte vom Typ `Bird` enthält!



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



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

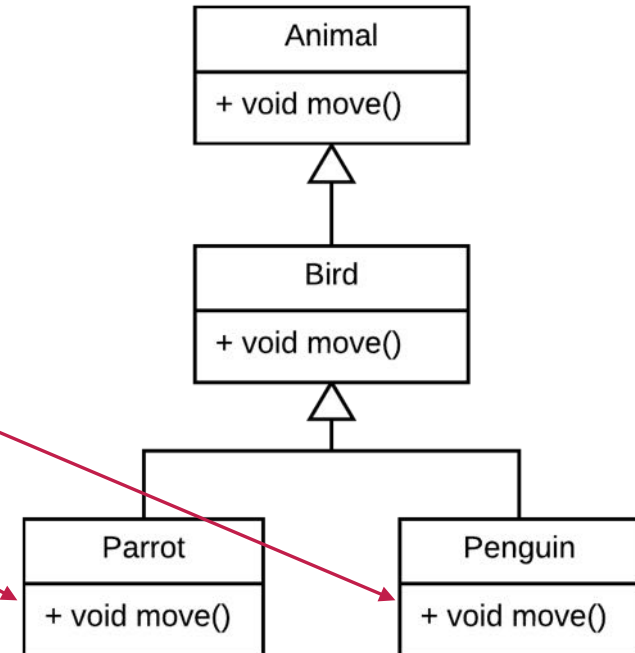
Ausgabe:

Ich fliege

Ich schwimme



```
[...]  
1   Bird parrot = new Parrot();  
2   Bird penguin = new Penguin();  
3  
4   penguin.move();  
5   parrot.move();  
[...]
```

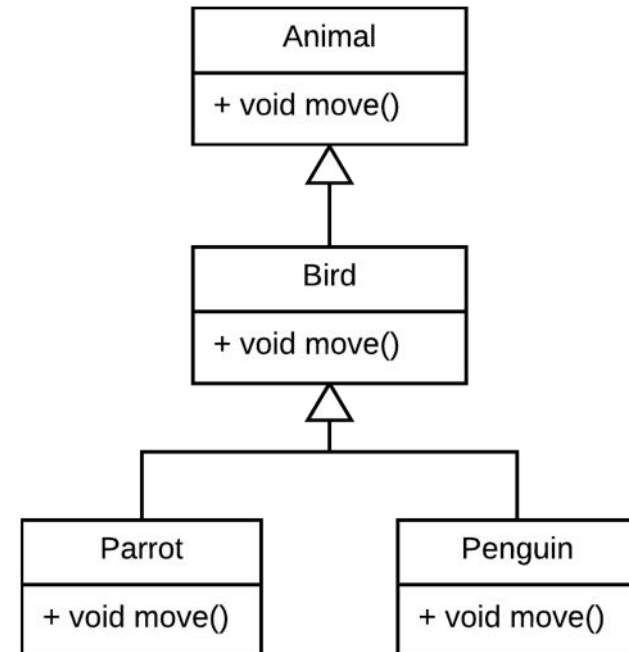


Vererbung

- Definiert ein "Protokoll" für eine Klassenfamilie:
 - Alle Subklassen besitzen alle geerbten Methoden und Attribute der Superklasse



```
[...]  
1   Bird parrot = new Parrot();  
2   Bird penguin = new Penguin();  
3  
4   penguin.move();  
5   parrot.move();  
[...]
```

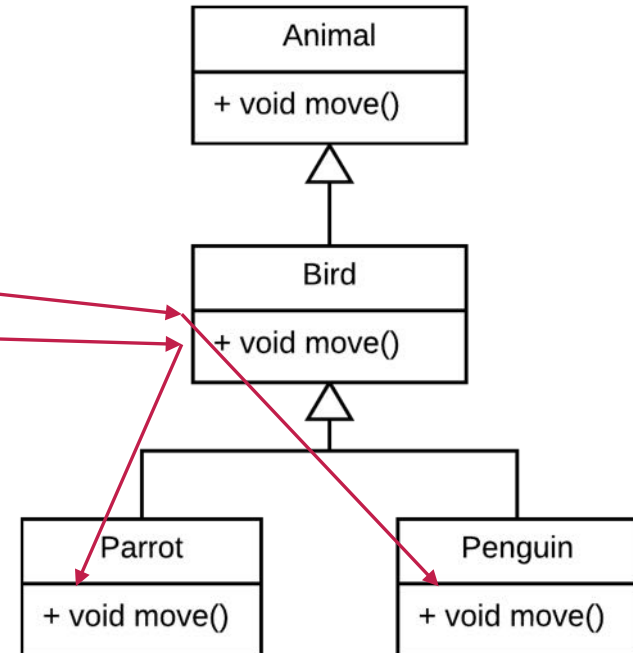


Polymorphie

- Einer Variablen können wir sowohl Objekte ihres Typs als auch aller ihrer Subtypen zuweisen
- Diese erfüllen per Definition unser Protokoll, daher können wir **immer** eine beliebige Subklasse anstelle der Superklasse einsetzen



```
[...]  
1   Bird parrot = new Parrot();  
2   Bird penguin = new Penguin();  
3  
4   penguin.move();  
5   parrot.move();  
[...]
```



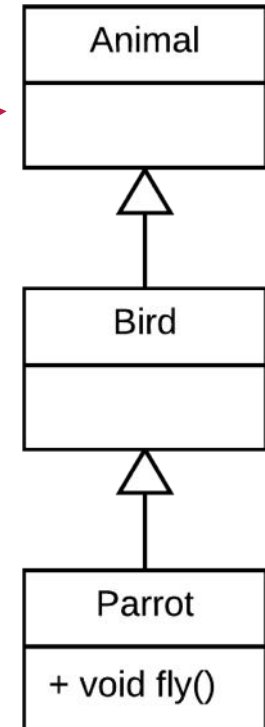
Ausgabe:

Ich schwimme
Ich fliege

Ein Vogel, der nicht mehr fliegen kann?



```
[...]  
1  Animal parrot = new Parrot();  
2  parrot.fly();  
[...]
```



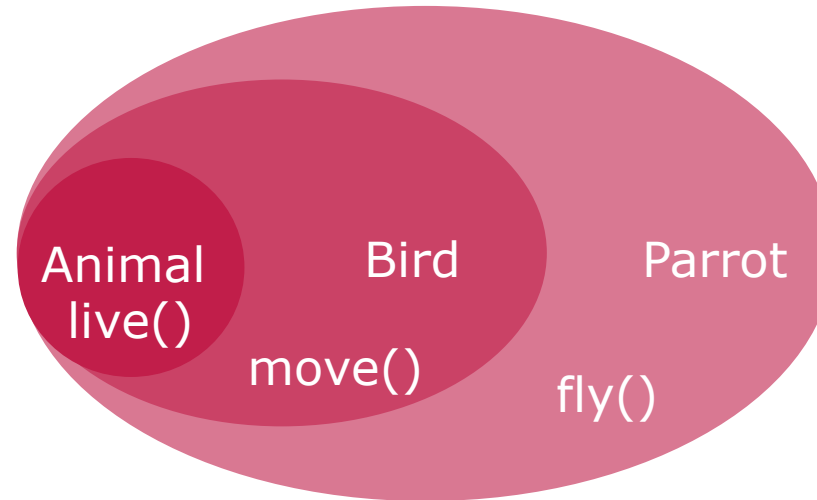
- Leider ein Fehler
- Java (der Compiler) entscheidet auf Grund des Datentyps ob man eine Methode aufrufen kann
 - Die Klasse `Animal` hat keine Methode `fly()`
 - Nur die Klasse `Parrot` implementiert die Methode



Polymorphie und Parameter

```
1 public class Robot {  
2  
3     // ...  
4     public void investigateAnimal(Animal animal) {  
5         // ...  
6     }  
7 }
```

- Die Methode `investigateAnimal()` erwartet ein Objekt der Klasse `Animal` als Parameter
- `investigateAnimal()` akzeptiert daher alle Subklassen von `Animal` als Argument
 - Neue Subklassen erfordern keine Anpassung des Codes innerhalb der Methode → bessere Wartbarkeit



- `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



this is super

openHPI-Java-Team
Hasso-Plattner-Institut



this.

```
1 public abstract class Bird {  
2     private String name;  
3     public Bird(String name) {  
4         this.name = name;  
5     }  
6     public String getName(){  
7         return name;  
8     }  
9 }
```

this.<attributBezeichner>

- erlaubt den Zugriff auf Attribute der eigenen Klasse
- Implizit gesetzt, wenn Bezeichner lokal nicht überschrieben wird



this()

```
1 public abstract class Bird {  
2     private String name;  
3  
4     public Bird() {  
5         this("Paco");  
6     }  
7  
8     public Bird(String name) {  
9         this.name = name;  
10    }  
11 }
```

this()

- ruft einen überladenen Konstruktor der gleichen Klasse auf
- Muss immer die erste Anweisung im Konstruktor sein
- Kann nur in Konstruktoren aufgerufen werden

super.



```
1 public class Bird {  
2  
3     public void move() {  
4         System.out.println("Ich bewege mich");  
5     }  
6 }
```

```
1 public class Parrot extends Bird {  
2     @Override  
3     public void move() {  
4         super.move();  
5         this.fly();  
6     }  
7     private void fly() {  
8         System.out.println("Ich fliege");  
9     }  
10 }
```




super()

```
1 public abstract class Bird {
2     private String name;
3
4     public Bird(String name) {
5         this.name = name;
6     }
7 }
8
9 public class Parrot extends Bird {
10     private String featherColor;
11
12     public Parrot(String name) {
13         super(name);
14         this.featherColor = "colorful";
15     }
16 }
```

- `super()` erlaubt den Aufruf des Konstruktors der Superklasse



super()

```
1 public class Parrot extends Bird {  
2     private String featherColor;  
3  
4     public Parrot(String name) {  
5         super(name);  
6         this.featherColor = "colorful";  
7     }  
8 }
```

super()

- Erlaubt den Aufruf des Konstruktors der Superklasse
- Muss immer die erste Anweisung im Konstruktor sein
- Kann nur in Kontruktoern aufgerufen werden



this();

Ruft einen überladenen Konstruktor der gleichen Klasse auf

Muss immer die erste Anweisung im Konstruktor sein

Kann nur in Konstruktoren aufgerufen werden

super();

Erlaubt den Aufruf des Konstruktors der Superklasse

Muss immer die erste Anweisung im Konstruktor sein

Kann nur in Konstruktoren aufgerufen werden

Achtung! Man kann nur entweder **this()** oder **super()** in einem Konstruktor gleichzeitig verwenden



this.<bezeichner>

Erlaubt Zugriff auf Attribute und Methoden der eigenen Klasse

super.<methodenBezeichner>

Erlaubt Zugriff auf Methoden der Superklasse



Deep Dive Java: Woche 3

openHPI-Java-Team

Hasso-Plattner-Institut

Ein kleiner Rückgriff auf Woche 1



```
1 public class Robot{  
2     private String name;  
3     //...  
4 }
```

Ein kleiner Rückgriff auch Woche 1



```
1 public class Story{
2     public static void main(String[] args){
3         Robot robin = new Robot();
4         System.out.println(robin.getName());
5     }
6 }
```

Ausgabe

null

Attribute bekommen default values.
Default value von String ist null!



- Schlüsselwort in Java
- Repräsentiert undefinierte Werte
- Vergleich auf `null` mit `!=` und `==`

Default values von Attributen



Datentyp	Default values (bei Attributen)
int	0
double	0.0
char	'\u0000'
boolean	false
String	null
Alle Objektdatentypen	null



```
1 public class Story{
2     public static void main(String[] args){
3         Robot robin = new Robot();
4         System.out.println(robin.getName().toLowerCase());
5     }
6 }
```

Exception in thread "main" java.lang.NullPointerException
at Woche3.story.main(story.java:4)



```
try {  
    // Quellcode, der eine Exception herbeiführen könnte  
} catch ( ... ) {  
    // Quellcode zum Behandeln der Exception  
}  
// ...
```

- Exception = wenn etwas "unerwartetes" passiert
- Sind Fehler, die bei der Ausführung des Programmes auftreten



```
1 public class Story{  
2     public static void main(String[] args){  
3         int i = "5";  
4     }  
5 }
```

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

Type mismatch: cannot convert from String to int

at Woche3.story.main(story.java:3)



```
1 public class Story{
2     public static void main(String[] args){
3         int[] array = new int[2];
4         for (int i = 0; i < array.length; i++) {
5             array[i] = 0;
6         }
7         System.out.println(array[2]);
8     }
9 }
```

Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 2
at Woche3.Story.main(Story.java:10)

- Wird geworfen (throw), wenn wir auch einen Index eines Arrays zugreifen wollen, der nicht existiert.



```
1 public class Story{
2     public static void main(String[] args) {
3         String name = null;
4         name = name.toLowerCase();
5     }
6 }
```

Exception in thread "main" java.lang.NullPointerException
at Woche3.story.main(story.java:4)

- Wird geworfen (throw) wenn eine Applikation versucht `null` zu verwenden, aber ein Objekt erwartet

