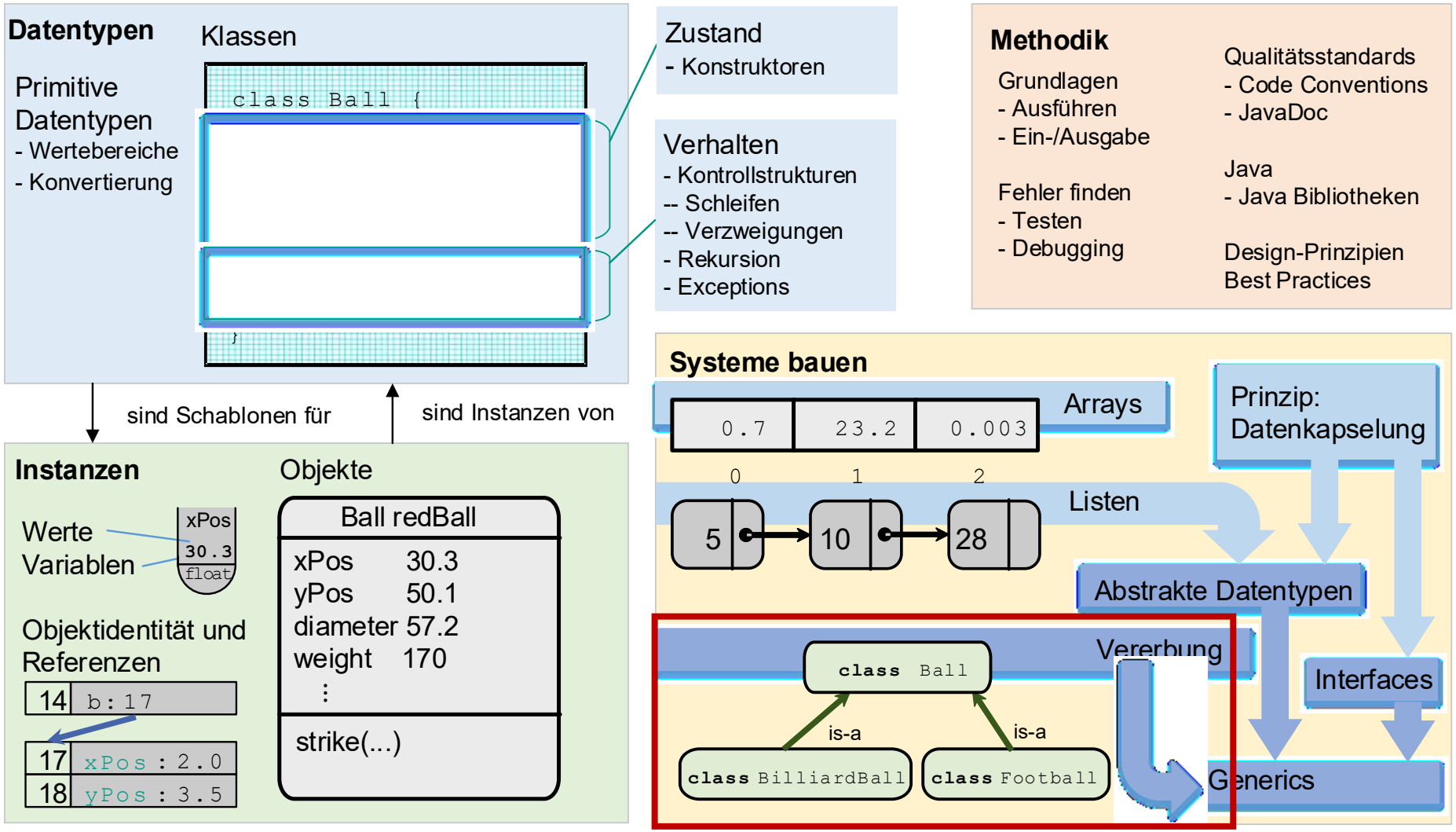
# Vorlesung Programmieren

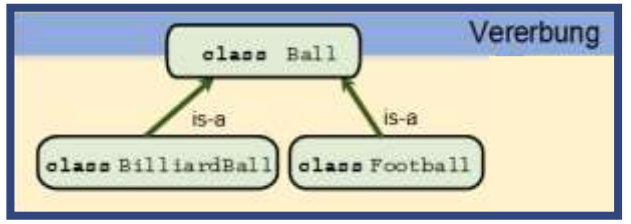
## 8. Vererbung

PD Dr. rer. nat. Robert Heinrich

# Vorlesungsüberblick: Objekt-orientiertes Programmieren in Java



# Lernziele




## Vererbung

- Sie können Klassen mithilfe von Vererbung erweitern
- Sie können Methoden überschreiben
- Sie wissen, wie dynamische Bindung in Java funktioniert und können damit erkennen, welche Methode jeweils aufgerufen wird
- Sie können durch Überschreiben polymorphe Methoden erstellen
- Sie können abstrakte Klassen definieren und verwenden

Quelle: http://phdcomics.com

# Einführendes Beispiel

- Wir wollen ein Java-Programm zur Verwaltung und Simulation von Lebewesen schreiben

- **Problem:**
  - Viel Wiederholungen
  - ➜ Kann zu Fehlern führen
  - ➜ Schlechte Wartbarkeit
  - ➜ Änderungen sind aufwendig

- Wie können wir diese Probleme vermeiden?

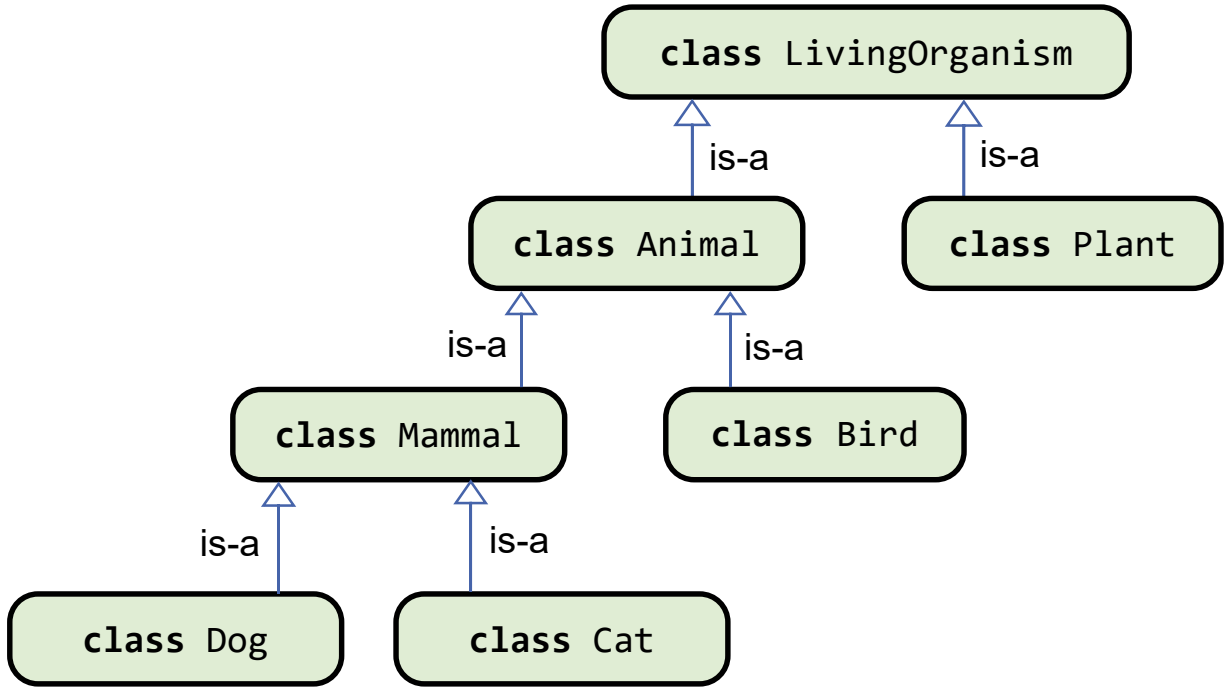
```
class Plant {
  private int age;
  private double weight;
  private double height;
  ...
}

class Dog {
  private String name;
  private int age;
  private double weight;
  private Date birthDate;
  ...
}

class Bird {
  private String name;
  private int age;
  private double weight;
  private Date hatchDate;
  private double wingspan;
  ...
}
```

# Konzept der Vererbung (I)

**Vererbung (*inheritance*, *subtyping*):**

- Mechanismus zur Implementierung von
  - Generalisierung bzw.
  - Spezialisierung von Typen/Klassen
- Modellierung einer "***is-a***"-Beziehung zwischen **Typen**

- **Vererbung ist keine Instanziierung (also Beziehung zwischen Typ und Instanz)**



# Konzept der Vererbung (II)

**Allgemein:**

- **Unterklasse (*subclass*)**: die speziellere Klasse (→ Subtyp)
- **Oberklasse (*superclass*)**: die generellere Klasse (→ Supertyp)

- Eine Oberklasse wird z.T. auch als **Basisklasse (*base class*)** oder **Elternklasse (*parent class*)** bezeichnet; eine Unterklasse als **abgeleitete Klasse (*derived class*)** oder **Kindklasse (*child class*)**

**Beispiele:**

- **`Bird`** ist (direkte) Unterklasse / Kindklasse von **`Animal`**
- **`LivingOrganism`** ist (direkte) Oberklasse / Elternklasse von **`Animal`**
- Auch indirekt: **`Cat`** ist Unterklasse von **`LivingOrganism`**

# Vererbung in Java

- **Schema:** `class Unterklasse extends Oberklasse { ... }`

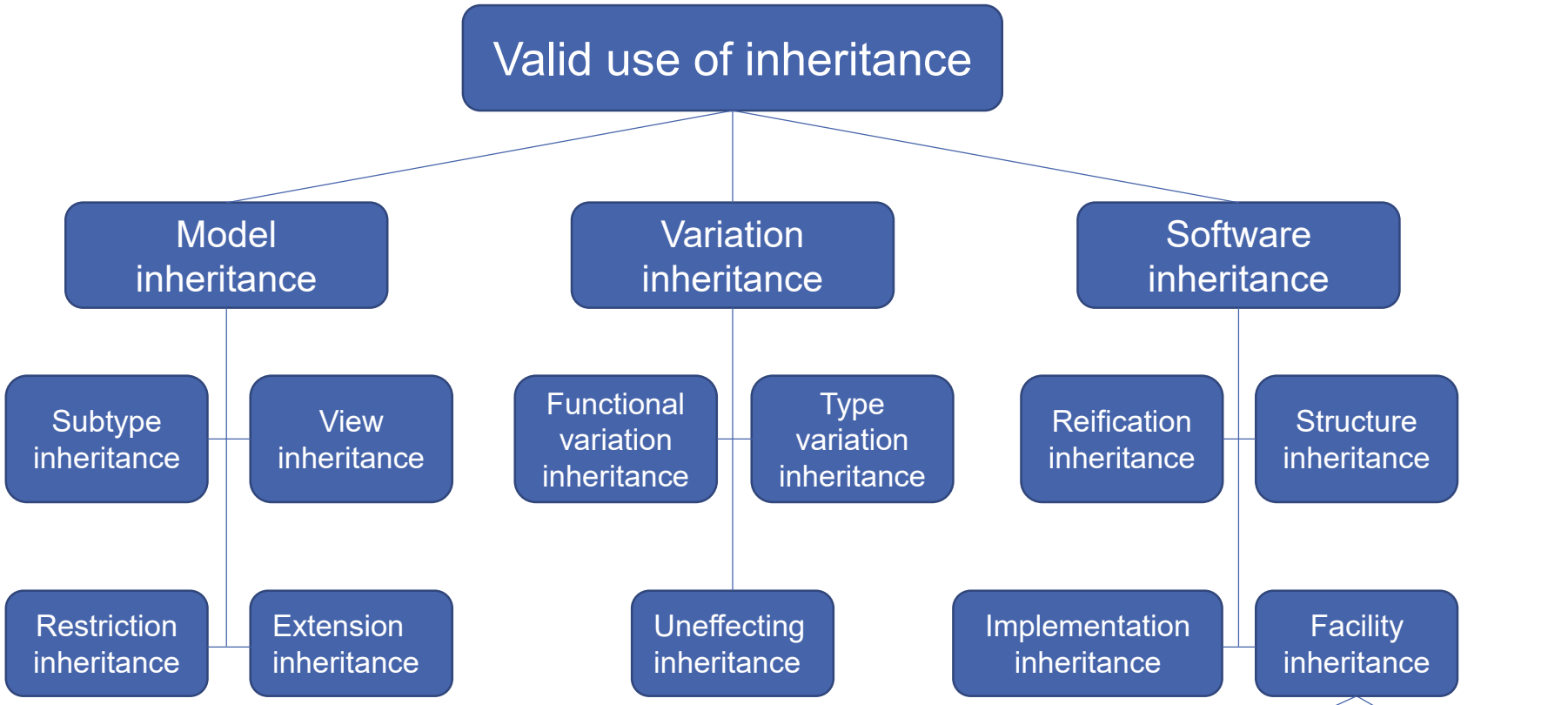
**Beispiele**

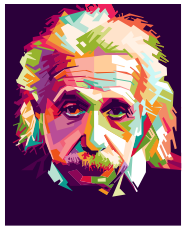
```
class Animal extends LivingOrganism { ... }
class Plant extends LivingOrganism { ... }
class Bird extends Animal { ... }
class Mammal extends Animal { ... }
class Dog extends Mammal { ... }
```

- **Vererbung wird verwendet zur:**
  - Spezialisierung (bei der objektorientierten Modellierung)
  - Substituierbarkeit (Ersetzbarkeit)
  - Wiederverwendung von Code ("code-inheritance")

Weitere Verwendungen: Bertrand Meyer. "The many faces of inheritance: A taxonomy of taxonomy." *Computer* 29.5 (1996): 105-108 und
Bertrand Meyer. Kapitel "A taxonomy of taxonomy„ in "Object-oriented software construction", 2. ed., 1997, Upper Saddle River, NJ : Prentice Hall PTR

# Verwendung von Vererbung



Bertrand Meyer. "The many faces of inheritance: A taxonomy of taxonomy." *Computer* 29.5 (1996): 105-108.
Bertrand Meyer "Object-oriented software construction", 2. ed., 1997, Upper Saddle River, NJ : Prentice Hall PTR

# Beziehung zwischen Ober- und Unterklasse

**Vererbung:** Unterklasse **erbt** (fast) alle Merkmale der Oberklasse

- Attribute
- Methoden
- Geschachtelte Klassen (*nested classes*)
- Nur prinzipiell in der Unterklasse sichtbare Merkmale werden vererbt
- *Keine* Vererbung von Konstruktoren

**Effekt:**

- Erweiterung: Unterklasse hat (in der Regel) mehr (speziellere) Eigenschaften (also Attribute und / oder Methoden) als Oberklasse
- *is-a*-Beziehung
- Substituierbarkeit: Unterklasse immer im Kontext der Oberklasse einsetzbar
  - "*Jede Katze ist ein Säugetier. Wer nur ein Säugetier braucht, muss auch mit einer Katze zufrieden sein.*"

# Beispiel Vererbung in Java

## Schlüsselwort: `extends`

## Beispiel:

- `Mammal` erweitert `Animal` um ein Attribut
- `Dog` fügt eine Methode zu `Mammal` hinzu

## Zuweisungen:

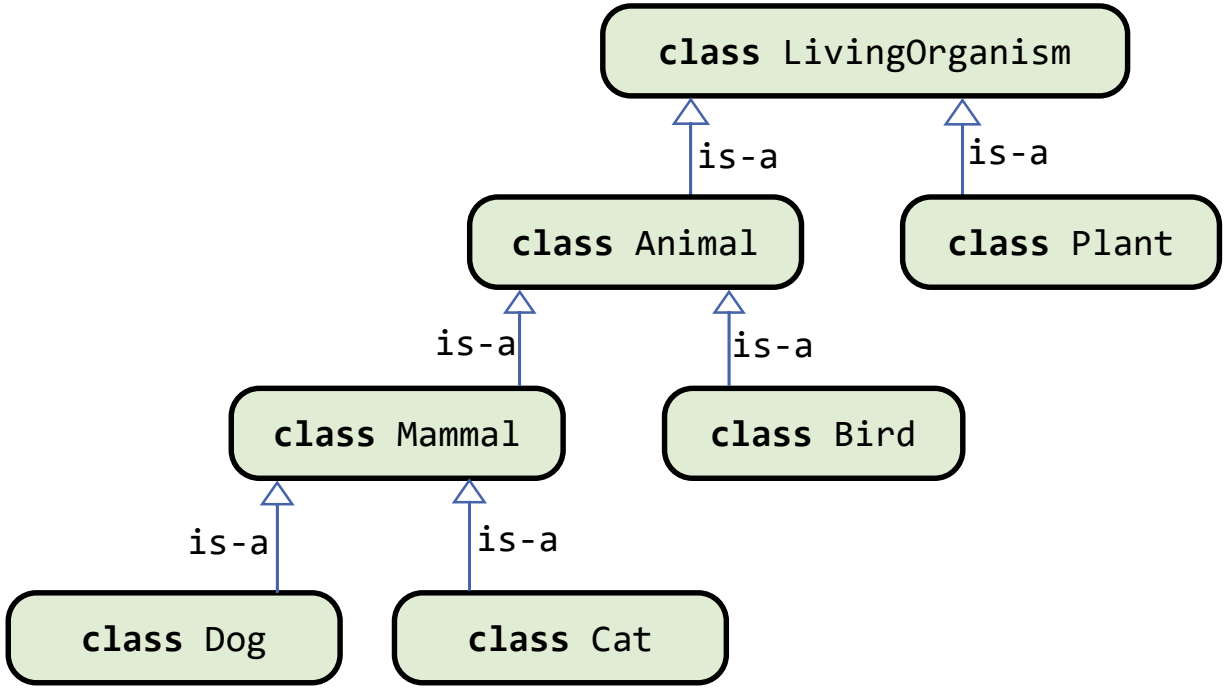
- **Unterklassen** an **Oberklasse** erlaubt,
- `Mammal m = new Dog();`
- Nicht umgekehrt
- `Dog d = new Mammal();`

(Alle Hunde sind Säugetiere, aber nicht alle Säugetiere sind Hunde) widening reference conversion

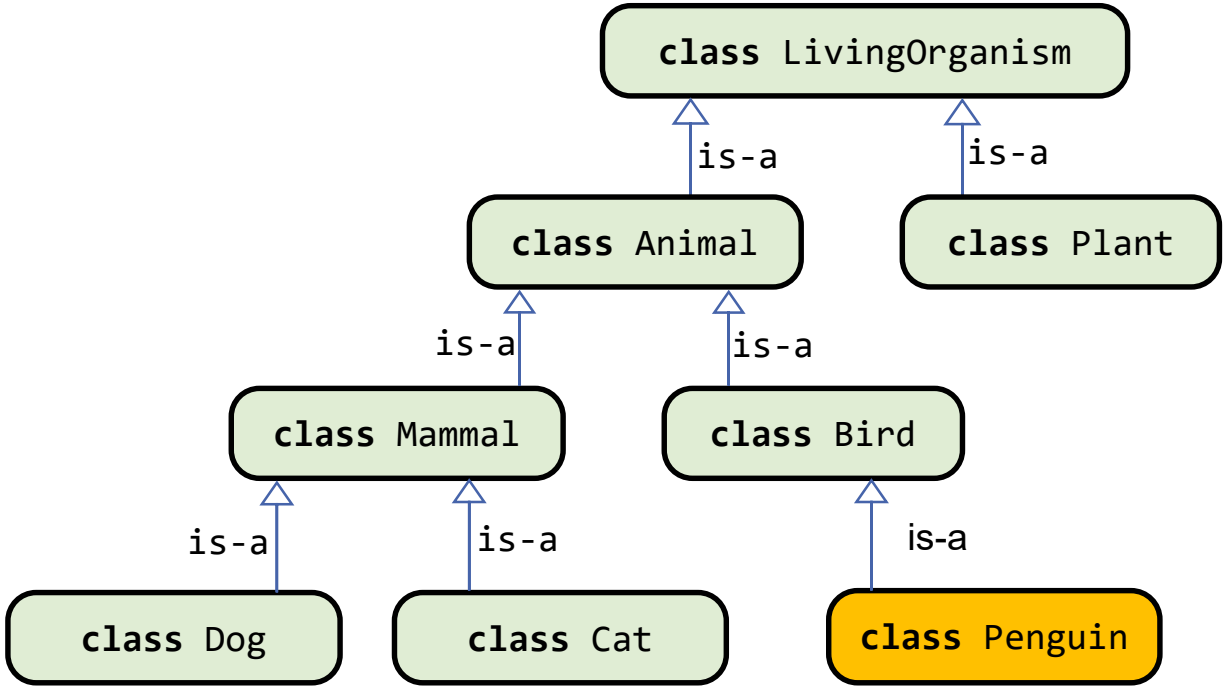
```
class Animal {
  void escape() { ... }
}

class Mammal extends Animal {
  Date birthday;
}

class Dog extends Mammal {
  void bark() { ... }
}

Dog fiffi = new Dog();
fiffi.bark();

Mammal m = fiffi;
Animal a = m;
a.escape();
```

# Jetzt sind Sie gefragt: Vererbung

- Wo würde ein Pinguin (`Penguin`) in unsere Vererbungshierarchie eingefügt werden?
- Wie drückt man es in Java aus?



# Lösung: Vererbung




```
class Penguin extends Bird {
  ...
}
```

# Überschreiben von Methoden

- **Anpassen von Verhalten bei Bedarf: Überschreiben einer Methode**
  - gleicher Methodenname
  - gleiche Parameteranzahl und Parametertypen
  - Rückgabetyp: Subtyp des vorherigen Rückgabetyps (oder gleich)
  - Annotation `@Override`
- Überschreiben von Attributen ist nicht möglich!

```
class Animal {
  void escape() { ... }
}

class Bird extends Animal {
  @Override
  void escape() {
    ...
    flyAway();
  }
  ...
}

class Penguin extends Bird {
  @Override
  void escape() {
    ...
    dive();
  }
  ...
}
```

# Dynamische Bindung

```
Penguin p = new Penguin();
Bird b1 = new Bird();
Bird b2 = p;
```

Welche Methode wird ausgeführt?

- `p.escape();`
  → Code aus der Klasse `Penguin`
- `b1.escape();`
  → Code aus der Klasse `Bird`
- `b2.escape();`
  → Code aus der Klasse `Penguin`

```
class Animal {
  void escape() { ... }
}

class Bird extends Animal {
  @Override
  void escape() {
    ...
    flyAway();
  }
  ...
}

class Penguin extends Bird {
  @Override
  void escape() {
    ...
    dive();
  }
  ...
}
```

# Dynamische Bindung – Definition

**Definition:** Ein Methodenaufruf `obj.m(...)` wird erst zur Laufzeit (*dynamisch*) an die passende Methodendefinition gebunden.

- Entscheidungsgrundlage ist der **Typ des von `obj` referenzierten Objekts** – **nicht der Typ der Variablen `obj`**.
- Z.B.: `Penguin p = new Penguin(); Bird b2 = p; b2.escape();`

**Unterscheidung**:

- **statischer Typ**: Typ bei Deklaration im Programmtext zur Entwurfszeit (`Bird`)
- **dynamischer Typ**: tatsächlicher Typ des Bezugsobjekts zur Laufzeit (`Penguin`)

**Bemerkung:**

- Die Parametertypen hingegen werden für die Entscheidung über die auszuführende Methode statisch berechnet.

**Vorteil:** Code ist einfacher erweiterbar und wartbar.

**Nachteil:** Minimaler Laufzeit-Overhead bei jedem Methodenaufruf.

# Dynamische Bindung – Beispiel (I)

```java
class Shape {
  double area() { ...
  // aufwendige Berechnung der
  // Fläche (z.B. Integral)
  ...
  }
}

class Circle extends Shape {
  double radius;

  Circle(double r) {
    radius = r;
  }

  @Override
  double area() {
    return Math.PI * radius * radius;
  }
}
```

```java
class Rectangle extends Shape {
  double width;
  double height;

  Rectangle(double width, double height) {
    this.width = width;
    this.height = height;
  }

  @Override
  double area() {
    return width * height;
  }
}
```

# Dynamische Bindung – Beispiel (II)

```
static double totalArea(Shape[] shapes) {
  double totalArea = 0.0;
  for (int i = 0; i < shapes.length; i++) {
    totalArea += shapes[i].area();
  }
  return totalArea;
}

Shape s1 = new Circle(10.0);
Shape s2 = new Rectangle(1.0, 4.5);
Shape[] shapes = {s1, s2, s1 };
double total = totalArea(shapes);
```

- Für jedes Element im Array wird dynamisch bestimmt, welche `area()`-Methode aufgerufen wird.
- **Vorteil:** Bei Erweiterung, z.B. um eine Klasse `Triangle`, muss die Methode `totalArea` nicht angepasst werden.

# Dynamische Bindung

- Überschreiben und Substituierbarkeit: Welche Methode wird ausgeführt?
- **Entscheidungsgrundlage:** Laufzeittyp des Bezugsobjekts `shapes[i]`
  - Beginnend bei diesem Typ:
    - Suche nach oben in Klassenhierarchie
    - Ausführen des ersten Treffers

→ **Dynamische Bindung** (*late binding*)

- Realisiert **Polymorphismus** (Vielgestaltigkeit)
  - Verschiedenes Verhalten kann beim Aufruf von Methoden über Variablen desselben statischen Typs ausgeführt werden
    - Beispiel eben: `b1.escape()` vs. `b2.escape()`
    - Unterschiedliches Verhalten, obwohl statischer Typ von `b1` und `b2` gleich
  - Verhalten ist abhängig vom Laufzeittyp des Bezugsobjekts

→ polymorphe Methode – Methode ist mehrfach implementiert

# Dynamische Bindung vs. Fallunterscheidung

- Ohne dynamische Bindung benötigt man Fallunterscheidung, um unterschiedliche Objekte gleichzeitig zu verwalten.
- Beispiel in MODULA-2:

```
PROCEDURE area(s: Shape): REAL
BEGIN
  CASE s.typeid OF
    Circle:
      RETURN PI * s.circle.radius * s.circle.radius;
    Rectangle:
      RETURN s.rectangle.width * s.rectangle.height;
...
END
```

➜ Bei Einführung eines neuen Shape-Typs müssen sämtliche Fallunterscheidungen überprüft und ggf. erweitert werden

# Jetzt sind Sie gefragt: Dynamische Bindung

- **Ziel**: Eine Methode `FeedTest.feedAll(Animal[] animals)`, die für ein gegebenes Array mit Tieren diese auf ihre Art und Weise fressen lässt (d.h. jeweils eine Methode `feed()` aufruft), ohne Fallunterscheidung
- Ein Aufruf soll so möglich sein

```
Penguin p = new Penguin(); Dog d = new Dog(); Cat c = new Cat();
Animal[] animals = {p, d, c};
FeedTest.feedAll(animals);
```

… und diese Ausgabe erzeugen:

```
C:\Users\Anne\Documents\lehre\WS1718_Programmieren\Demo\Animals>java FeedTest
Penguin eats fish
Dog eats dog food
Cat eats cat food
```

- (1) Welche Klassen müssen Sie dazu erweitern? Um welche Methoden?
- (2) Wie realisieren Sie `FeedTest.feedAll(Animal[] animals)`?

# Dynamische Bindung: Lösung

```java
class Dog extends Mammal {
  ...
  @Override
  void feed(){
    System.out.println("Dog eats dog food");
  }
}
```

`Cat` und `Penguin` analog

```java
class Animal extends LivingOrganism {
  ...
  void feed() { /* do something */ }
}
```

Unschön, da hier keine sinnvolle Implementierung möglich
→ Abstrakte Klassen

`Mammal` und `Bird` müssen nicht erweitert werden

```java
public static void feedAll(Animal[] animals){
  for (Animal animal : animals){
    animal.feed();
  }
}
```

# Überschreiben von Attributen

- **Das Überschreiben von Attributen ist nicht möglich.**
  Ein Attribut verschattet ein gleichnamiges Attribut aus der Oberklasse.
- Hintergrund: ein Objekt soll immer auch die Rolle von Objekten der Oberklassen spielen können.

```
class Mammal {
  int age = 0;

  void setAge(int age) {
    this.age = age;
  }
}

class Dog extends Mammal {
  int age = 0;

  void setDogAge(int age) {
    this.age = age;
  }
}
```

**Schlechter Stil:** Attribute in Vererbungshierarchie sollten nicht gleich heißen

```
Dog d = new Dog();
d.setAge(3);
d.setDogAge(21);
System.out.println(d.age);
```

➔ Ausgabe: "21"

```
Mammal dog = d;
System.out.println(dog.age);
```

Was wird ausgegeben?

➔ "3"

# Das Schlüsselwort super

- Zugriff auf Attribute und Methoden der Oberklasse
- Dynamische Bindung bei Methoden einmalig aufgehoben
- Analogon zu **`this`**

```
class Dog extends Mammal {
  int age = 0;

  void setDogAge(int age) {
    this.age = age;    // Attribut age aus Klasse Dog
  }

  void setMammalAge(int age) {
    super.age = age;  // Attribut age aus Klasse Mammal
  }
}
```

# Konstruktoren

- Konstruktoren werden **nicht** vererbt
- In einem Konstruktor wird **immer** zunächst ein Konstruktor der Oberklasse aufgerufen
  - Dies geschieht mittels der Anweisung `super(...)`
  - `super(...)` ist immer die erste Anweisung in einem Konstruktor
  - Wird `super(...)` nicht explizit aufgerufen, so wird von Java implizit `super()` (ohne Parameter) ausgeführt
- Wird in einer Klasse kein Konstruktor implementiert, so fügt Java einen Default-Konstruktor ein
  - Z.B. für die Klasse Dog: `public Dog() { super(); }`
- Beim Aufruf von `super()` muss die direkte Oberklasse eine Default-Konstruktor oder einen Konstruktor ohne Parameter besitzen
- Gibt es in einer Klasse **keinen** Konstruktor ohne Parameter, so müssen in allen Subklassen Konstruktoren definiert werden

# Sichtbarkeit und Modifier

- Im Zusammenhang mit Vererbung gibt es einen weiteren Modifikator zur Einschränkung der Sichtbarkeit: `protected`

| Modifikator | Sichtbarkeit | | | |
|---|---|---|---|---|
| | in Klasse | in Paket | in Unterklassen | „Welt" |
| `public` | ✓ | ✓ | ✓ | ✓ |
| `protected` | ✓ | ✓ | ✓ | – |
| – | ✓ | ✓ | –[1] | – |
| `private` | ✓ | – | – | – |

[1]: außer, wenn Unterklasse im gleichen Paket

→ Mittel zur Realisierung des Geheimnisprinzips

# Jetzt sind Sie gefragt: Attribute und Methoden

## Erbt eine Unterklasse sowohl Attribute als auch Methoden?

(A) Nein, nur Attribute werden vererbt
(B) Nein, nur Methoden werden vererbt
(C) Ja, sämtliche Variablen und Methoden werden vererbt
(D) Ja, aber `private` Methoden und Attribute werden nicht vererbt

# instanceof

- Der `instanceof`-Operator prüft, ob ein gegebenes Objekt den gegebenen Typ haben kann
- **Syntax:** `Objekt instanceof Klasse`

**Beispiel**

```
static void isDog(Mammal m) {
  if (m instanceof Dog) {
    System.out.println("Mammal is a dog");
    } else {
      System.out.println("Mammal is not a dog");
    }
  }
}
```

```
isDog(new Cat());
isDog(new Dog());
isDog(new Mammal());
isDog(null);
```

```
// → "Mammal is not a dog"
// → "Mammal is a dog"
// → "Mammal is not a dog"
// → "Mammal is not a dog"
```

# Typ-Umwandlungen (*type casts*)

- Sind wir uns sicher, dass wir es mit einer Katze zu tun haben, so kann diese auch miauen

```
void catMeow(Mammal m) {
  if (m instanceof Cat) {
    Cat c = (Cat) m;
    c.meow();
  }
}
```

- Ein Typecast ändert den statischen Typ eines Objekts

→ Eine Katze ist eine Katze, kann aber auch als Säugetier behandelt werden – nicht jedoch als Hund

- **Syntax der Typ-Umwandlung:**

```
(Typname) Variable
```

# Up- und Down-Casts

**Up-Cast** bezeichnet eine Typumwandlung auf eine ***Superklasse***

- ist harmlos (→ *is-a*-Beziehung)
- wird implizit bei Zuweisungen und Methodenaufrufen ausgeführt

**Down-Cast** bezeichnet eine Typumwandlung auf eine ***Subklasse***

- ist gefährlich (kann zu einem Laufzeitfehler führen)
- muss immer explizit angegeben werden
- sollte immer vorher mit **`instanceof`** geprüft werden

# Die Klasse `Object`

- Die **Klasse `Object`** (aus dem Paket `java.lang`) ist implizit Oberklasse jeder anderen Klasse. Sie bildet die **Wurzel der Klassenhierarchie**.

- Einige Methoden in `Object`:
  - `equals(Object obj)`: Vergleicht das aktuelle Objekt mit `obj`
  - `toString()`: Liefert eine String-Repräsentation des aktuellen Objekts
  - `clone()`: Liefert eine 1:1-Kopie des aktuellen Objekts

- Diese Methoden müssen bei Bedarf überschrieben werden!


DSiS – Dependability of Software-intensive Systems Group
KASTEL – Institute of Information Security and Dependability

# Inhaltliche Gleichheit: `equals(Object obj)`

- **API-Spezifikation von `Object.equals(Object obj)`:**
- Indicates whether some other object is „equal to" this one.
- The `equals` method implements an **equivalence relation** on non-null object references:
  - It is reflexive: for any non-null reference value `x`, `x.equals(x)` should return **`true`**.
  - It is symmetric: for any non-null reference values `x` and `y`, `x.equals(y)` should return **`true`** if and only if `y.equals(x)` returns **`true`**.
  - It is transitive: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns **`true`** and `y.equals(z)` returns **`true`**, then `x.equals(z)` should return **`true`**.
  - It is consistent: for any non-null reference values `x` and `y`, multiple invocations `x.equals(y)` consistently return **`true`** or consistently return **`false`**, provided no information used in `equals` comparisons on the objects is modified.
  - For any non-null reference value `x`, `x.equals(null)` should return **`false`**.
- The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns **`true`** if and only if x and y refer to the same object (`x == y` has the value **`true`**).

https://docs.oracle.com/en/java/javase/15/docs/api/index.html

# Inhaltliche Gleichheit

Beim Überschreiben von **`equals`** müssen die Eigenschaften Reflexivität, Symmetrie, Transitivität und Konsistenz auf jeden Fall erhalten bleiben.

**Checkliste** für `equals(Object obj)`:

- **Syntax**: Ist der Rückgabetyp **`boolean`** und der Typ des Parameters `Object`?
- **Reflexivität**: Liefert die Methode **`true`**, wenn für den Parameter `obj` **`this`** eingesetzt wird?
- **Symmetrie**: Kann in der Implementierung **`this`** und `obj` ausgetauscht werden, ohne dass ein semantischer Unterschied entsteht?
- **Transitivität**: Abhängig vom Einzelfall. Bei Gleitkommazahlen z.B. keinen ε-Umgebungstest in `equals` verwenden.
- **Konsistenz**: Die Methode darf den (für die Gleichheit relevanten Teil des) Zustand(s) des Objekts nicht verändern.

# Grenzen der Vererbbarkeit: `final`

**Schon bekannte Verwendungen von `final`:**

- **Attribut**: konstanter Wert während Lebensdauer des Objekts
- **Parameter**: unveränderlich im Methodenrumpf

**Neu im Zusammenhang mit Vererbung:**

- **Klasse**: Keine Unterklassen möglich
  - **Gründe:** Sicherheit, Performanz
  - **Beispiele:** `java.lang.System`, `java.lang.String`
- **Methode**: Kein Überschreiben möglich
  - Verhalten fixiert

# Abstrakte Klassen

- Oberklasse ohne instanziierte Objekte (kein **`new`**)
- Deklaration abstrakter Methoden (**`abstract`**):
  - Keine oder unvollständige Implementierung; fehlende Implementierung muss in Unterklassen hinzugefügt werden
  - z.B. Methode `abstract double area();` der Klasse `Shape`
- Attribute, Konstruktoren und Methoden wie normale Klassen
  → Verwendung abstrakter Methoden mit dynamischer Bindung
- **Syntax:** `abstract class Klassenname { ... }`
  In der Klasse können einzelne Methoden als **`abstract`** deklariert sein; diese enthalten dann keine Implementierung

**Beispiel**

```
abstract class Mammal {
  Date birthday;
  void giveBirth() { ... }
  abstract void speak();
}
```

```
class Cat extends Mammal {
  void meow() { ... }
  @Override
  void speak() { meow(); }
}
```

Es gibt kein reines Säugetier, nur konkrete Unterklassen, z.B. Katzen

# Jetzt sind Sie gefragt: Abstrakte Klassen

**Kann eine abstrakte Klasse sowohl abstrakte als auch nicht-abstrakte Methoden enthalten?**

(A) Nein, alle Methoden müssen als abstrakt deklariert sein
(B) Nein, keine der Methoden darf als abstrakt deklariert sein
(C) Ja, aber abstrakte Methoden werden nicht vererbt
(D) Ja, und sämtliche Methoden werden vererbt
(E) Ja, aber abstrakte Methoden dürfen nicht überschrieben werden

# Jetzt sind Sie gefragt: Vorteile der Vererbung

**Was ist *kein* Vorteil der Verwendung von Vererbung?**

(A) Code, der von mehreren Klassen verwendet wird, muss nur ein Mal geschrieben werden

(B) Die Erweiterung um neue Programmeigenschaften wird vereinfacht

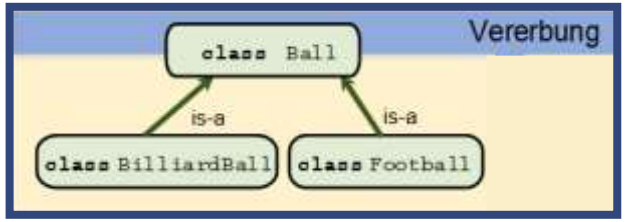
(C) Programmcode kann einfacher wiederverwendet werden

(D) Abgeleitete Klassen sind unabhängig von der Basisklasse

(E) Programmcode ist besser strukturiert

# Zusammenfassung

- Vererbung: “Ist-ein” Beziehung zwischen Klassen
  - Abbildung der Klassenhierarchien in der realen Welt
  - Weniger Code-Wiederholungen
  - Gemeinsames einmal behandeln, gleichzeitig Spezialisierung zulassen

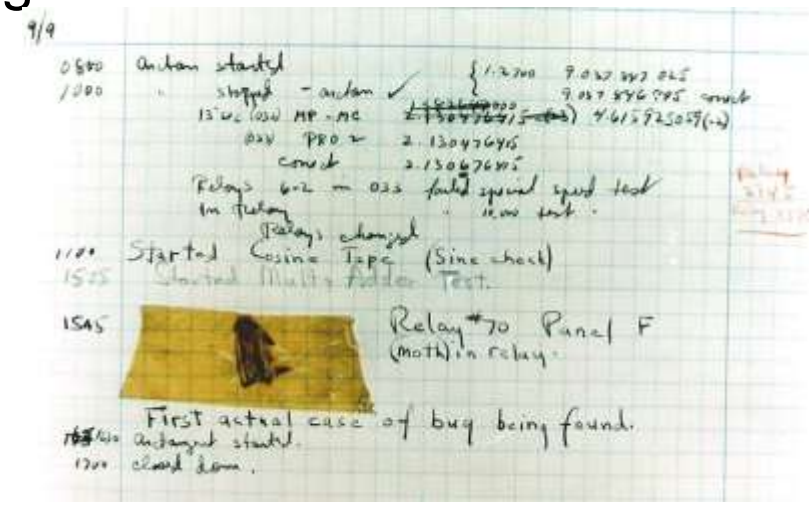



- Vererbt werden Methoden, Attribute, geschachtelte Klassen

- Dynamisches Binden statt Fallunterscheidung
  - Zur Klasse gehörende Logik wird gekapselt in Klasse
  - Bessere Wartbarkeit

# Debugging

- Der systematische Prozess der Fehlersuche und -behebung im Programm wird **Debugging** genannt.
- Hierzu finden Sie den Ilias-Wiki-Artikel „Debugging“
- Eine praktische Einführung in Debugging-Methoden erhalten Sie in Ihrem jeweiligen Tutorium im neuen Jahr




First actual case of a bug being found

Wikipedia: A page from the Harvard Mark II electromechanical computer's log, featuring a dead moth that was removed from the device https://en.wikipedia.org/wiki/Software_bug#/media/File:H96566k.jpg, Collection of the Smithsonian National Museum of American History

# Literaturhinweis - Weiterlesen

- Dietmar Ratz, Jens Scheffler, Detlef Seese und Jan Wiesenberger "Grundkurs Programmieren in Java", 7. Auflage, 2014 (mit Java 8), Hansa-Verlag
  - „Vererbung und Polymorphismus“
- Bertrand Meyer "Object-oriented software construction", 2. ed., 1997, Upper Saddle River, NJ : Prentice Hall PTR

# Prinzipien des Objekt-Orientierten Entwurfs

- OCP: Open-Closed-Principle (Bertrand Meyer)
- LSP: Liskov's Substitution Principle (Barbara Liskov)