CARL VON OSSIETZKY universität OLDENBURG

Sommersemester 2020

Objektorientierte Modellierung und Programmierung

Dr. Christian Schönberg

CARL VON OSSIETZKY universität OLDENBURG

Polymorphie

CARL VON OSSIETZKY universität OLDENBURG

Lernziele

- Überladen
- Überschreiben
- Dynamisches Binden
- Vererbung

CARL VON OSSIETZKY universität OLDENBURG

Überladen von Methoden

CARL VON OSSIETZKY universität OLDENBURG

Überladen von Operatoren

```
int x = 3 + 5; // 8
```

Parameter vom Typ **`int`**:
+ ist definiert als Integer-Addition.

```
String s = "3" + "5"; // "35"
```

Parameter vom Typ **`String`**:
+ ist definiert als String-Verkettung.

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Überladen von Methoden

```
public class Util {

    public static int plus(int x, int y) {
        return x + y;
    }

    public static String plus(String a, String b) {
        return a + b;
    }

}
```

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Überladen von Methoden

```
public class Util {

    public static int plus(int x, int y) {
        return x + y;
    }

    public static String plus(String a, String b) {
        return a + b;
    }

}
```

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Überladen von Methoden

```
public class Util {

    public static int plus(int x, int y) {
        return x + y;
    }

    public static String plus(String a, String b) {
        return a + b;
    }

}
```

CARL VON OSSIETZKY universität OLDENBURG

Überladen von Methoden

- Gleicher Methodenname
- Unterschiedliche Parameter(-zahl) oder -typen
- Ggf. unterschiedlicher Rückgabetyp
- Andere Implementierung
  - sollte aber strukturell vergleichbar sein

➢ Prinzip gleicher Name, analoges Verhalten

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Überladen von Methoden (2)

```
System.out.println(1);
System.out.println(3.14);
System.out.println("Hello");
System.out.println('x');
System.out.println(new Domino(7));
System.out.println();
```

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Überladen von Methoden (3)

```
public class Domino {

    private int dots;
    private boolean fallen;
    private Domino neighbor;

    public Domino() {
        Random rand = new Random();
        this.dots = rand.nextInt(12) + 1;
    }

    public Domino(int dots) {
        this.dots = dots;
    }
```

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Überladen von Methoden (3)

```
    public void place() {
        this.neighbor = null;
        fallen = false;
    }

    public void place(Integer dots) {
        place(new Domino(dots));
    }

    public void place(Domino neighbor) {
        this.neighbor = neighbor;
        fallen = false;
    }

}
```

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Überladen von Methoden (3)

```
Domino first  = new Domino(12);
Domino second = new Domino(8);
Domino third  = new Domino(9);
Domino fourth = new Domino(4);

first.place(second);
second.place(8);
third.place();
fourth.place(null);
```

Beispiel: Überladen von Methoden (3)

```
Domino first  = new Domino(12);
Domino second = new Domino(8);
Domino third  = new Domino(9);
Domino fourth = new Domino(4);

first.place(second);
second.place(8);
third.place();
fourth.place(null);
```

The method **place(Integer)** is ambiguous for the type **Domino**.

Beispiel: Überladen von Methoden (3)

```
Domino first  = new Domino(12);
Domino second = new Domino(8);
Domino third  = new Domino(9);
Domino fourth = new Domino(4);

first.place(second);
second.place(8);
third.place();
fourth.place(null);
```

The method **`place(Integer)`** is ambiguous for the type **`Domino`**.

`null` kann sowohl vom Typ **`Domino`** als auch vom Typ **`Integer`** sein.
Damit ist keine eindeutige Auswahl der Methode möglich!

CARL VON OSSIETZKY universität OLDENBURG

Überladen und Vererbung

```java
public class Fruit {

    private int weight;
    private int price;
    private String origin;

    public boolean isAvailable() {
        return true;
    }

}
```

```java
public class Banana extends Fruit {

    private int count;

    public boolean isAvailable(int bundle) {
        return count == bundle;
    }

}
```

CARL VON OSSIETZKY universität OLDENBURG

Vorteile durch Überladen

- Einfache Fallunterscheidung für verschiedene Typen als Parameter
  - z.B. **`int`** oder **`String`**, **`Apple`** oder **`Banana`**
- Gleiches bzw. ähnliches Verhalten für verschiedene Parameter: gleicher Name → analoges Verhalten

CARL VON OSSIETZKY universität OLDENBURG

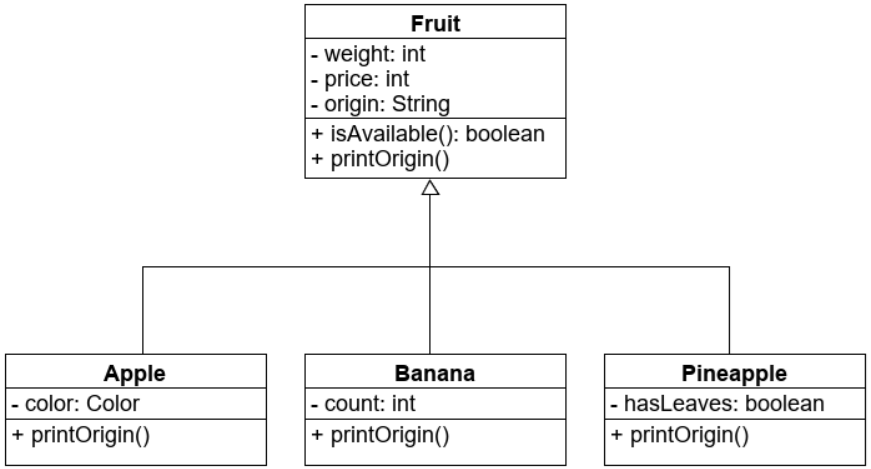
Polymorphie

Polymorphie

- Die Vererbungsbeziehung ist eine is-a Beziehung
  → überall, wo eine Instanz der Oberklasse erwartet wird, kann auch eine Instanz einer Unterklasse verwendet werden
  → die Oberklasse kann unterschiedliche Gestalten annehmen, sie ist polymorph
- Beispiel
  - einer Referenzvariablen vom Typ **`Fruit`** kann ein Objekt vom Typ **`Apple`** oder **`Banana`** zugewiesen werden
  - aber: über diese Variable kann dann nur auf die Attribute und Methoden von **`Fruit`** zugegriffen werden, nicht auf die von **`Apple`** oder **`Banana`**
  - die Variable, über die auf ein Objekt zugegriffen wird, definiert, welche Attribute und Methoden sichtbar sind

Beispiel: Polymorphie

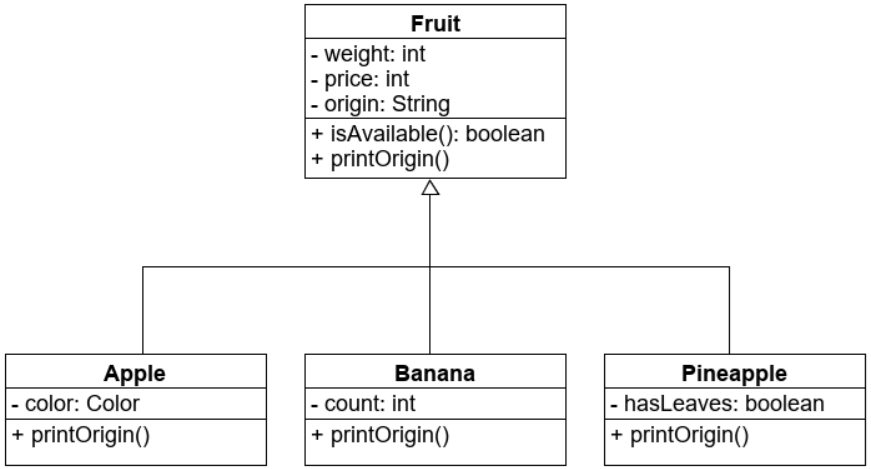
```
public class FruitBasket2 {

    public static void main(String[] args) {
        Fruit fruit = new Apple();
        System.out.println(fruit.getOrigin());
        System.out.println(fruit.getColor());

        Fruit[] basket = new Fruit[3];
        basket[0] = new Apple();
        basket[1] = new Banana();
        basket[2] = new Pineapple();
        check(basket);
    }

    private static void check(Fruit[] basket) {
        for (Fruit fruit : basket) {
            System.out.println(fruit.getWeight());
        }
    }

}
```



Beispiel: Polymorphie

```
public class FruitBasket2 {

    public static void main(String[] args) {
        Fruit fruit = new Apple();
        System.out.println(fruit.getOrigin());
        System.out.println(fruit.getColor());
        Fruit[] basket = new Fruit[3];
        basket[0] = new Apple();
        basket[1] = new Banana();
        basket[2] = new Pineapple();
        check(basket);
    }

    private static void check(Fruit[] basket) {
        for (Fruit fruit : basket) {
            System.out.println(fruit.getWeight());
        }
    }

}
```

The method **getColor()** is undefined for the type **Fruit**



Polymorphie (2)

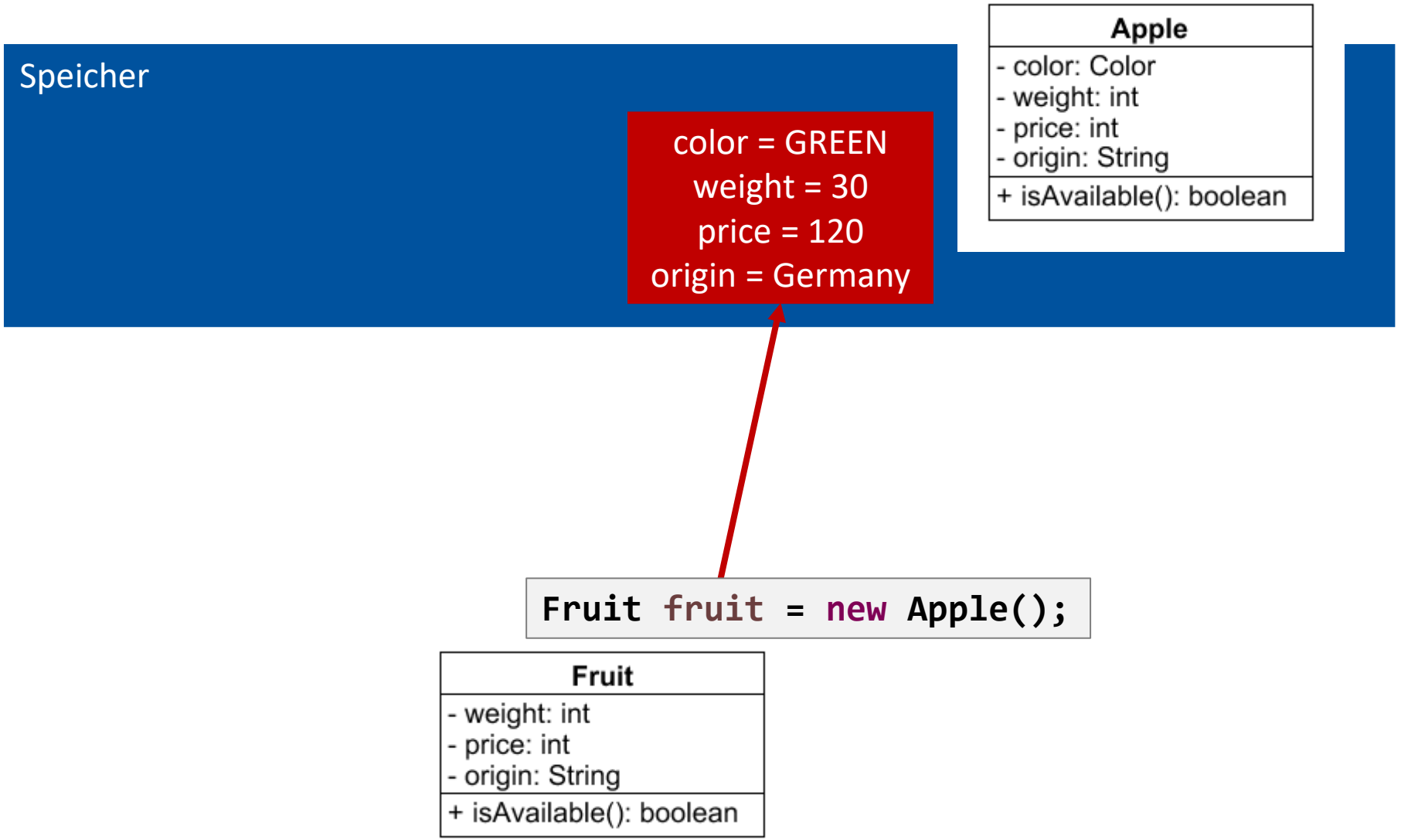
- Der Typ der Referenzvariablen, über den auf ein Objekt zugegriffen wird, entscheidet welche Attribute und Methoden sichtbar sind
  → Schnittstelle nach außen
  - ist die Variable vom gleichen Typ wie die Instanz, so kann über alle Attribute und Methoden der Instanz auf das Objekt zugegriffen werden
  - ist die Variable vom Typ einer Oberklasse der tatsächlichen Instanz, so kann nur über die Attribute und Methoden der Oberklasse auf das Objekt zugegriffen werden
    → falls ein Zugriff auf die Attribute oder Methoden der Unterklasse nötig ist **und** der Typ der Instanz bekannt ist: Type-Cast

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Polymorphie (2)

```
Fruit fruit = new Apple();
System.out.println(fruit.getOrigin());

Apple apple = (Apple) fruit;
System.out.println(apple.getColor());

System.out.println(((Apple) fruit).getColor());
```

Abbildung im Speicher



CARL VON OSSIETZKY universität OLDENBURG

Überschreiben

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Überschreiben von Methoden

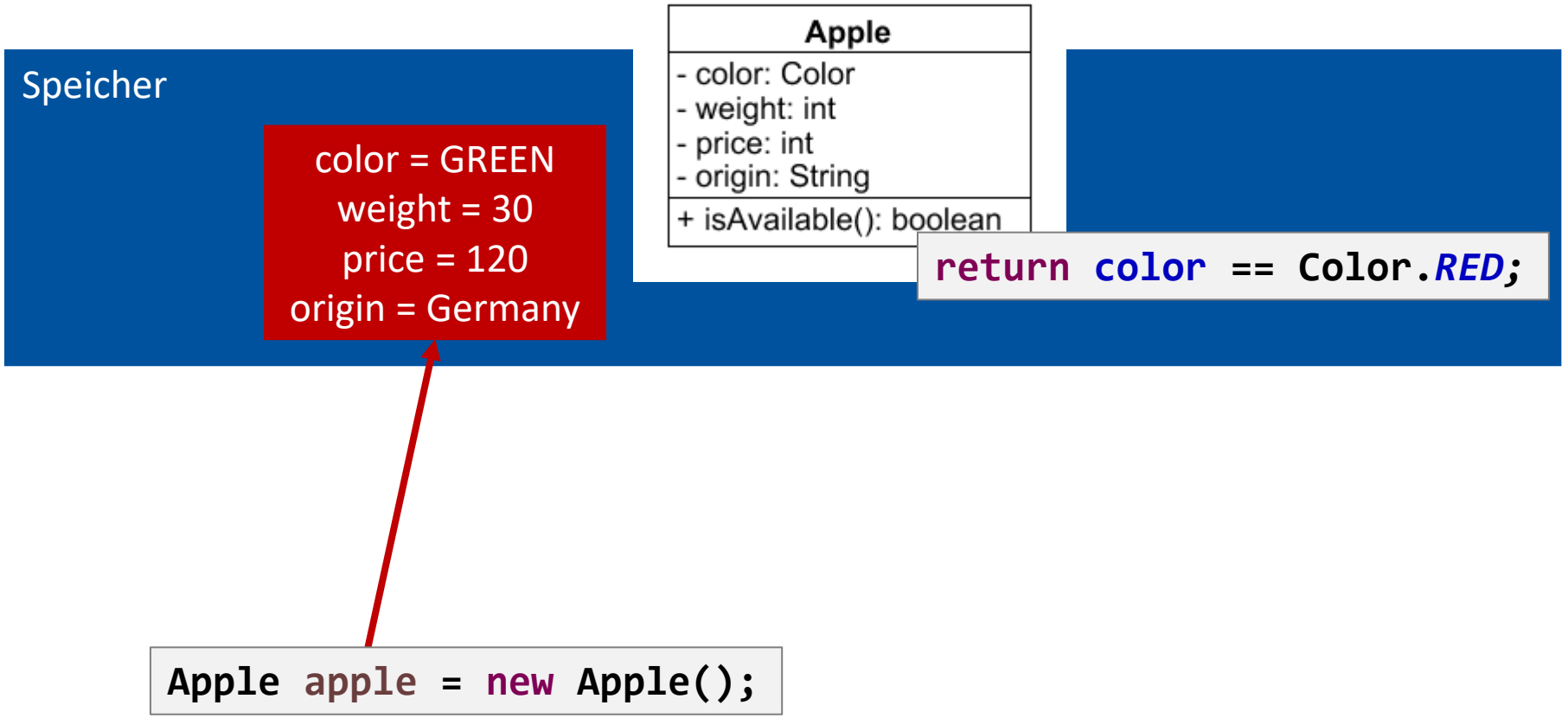
```java
public class Fruit {

    private int weight;
    private int price;
    private String origin;

    public boolean isAvailable() {
        return true;
    }

}
```

```java
public class Apple extends Fruit {

    private Color color;

    @Override
    public boolean isAvailable() {
        // only red apples available
        return color == Color.RED;
    }

}
```

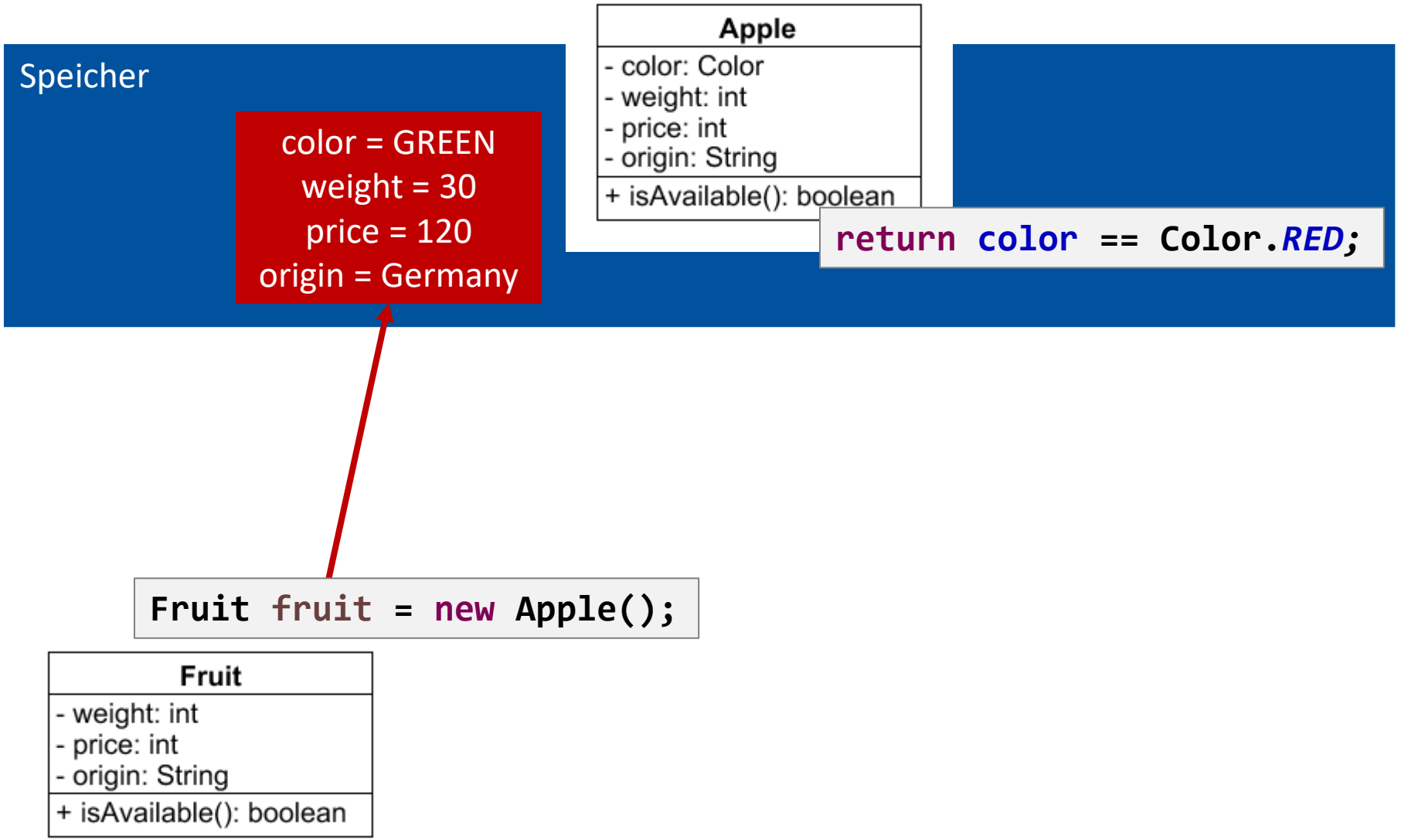
Überschreiben von Methoden

- Eine Oberklasse definiert eine Methode
- Eine Unterklasse definiert eine Methode mit exakt gleicher Signatur (Name, Parameter, Rückgabewert) aber verändertem Verhalten
  - das Verhalten kann in der spezielleren Unterklasse verfeinert werden
- Die Unterklasse wird instanziiert
  - → die veränderte Methode der Unterklasse wird aufgerufen
- Die Oberklasse wird instanziiert
  - → die unveränderte Methode der Oberklasse wird aufgerufen

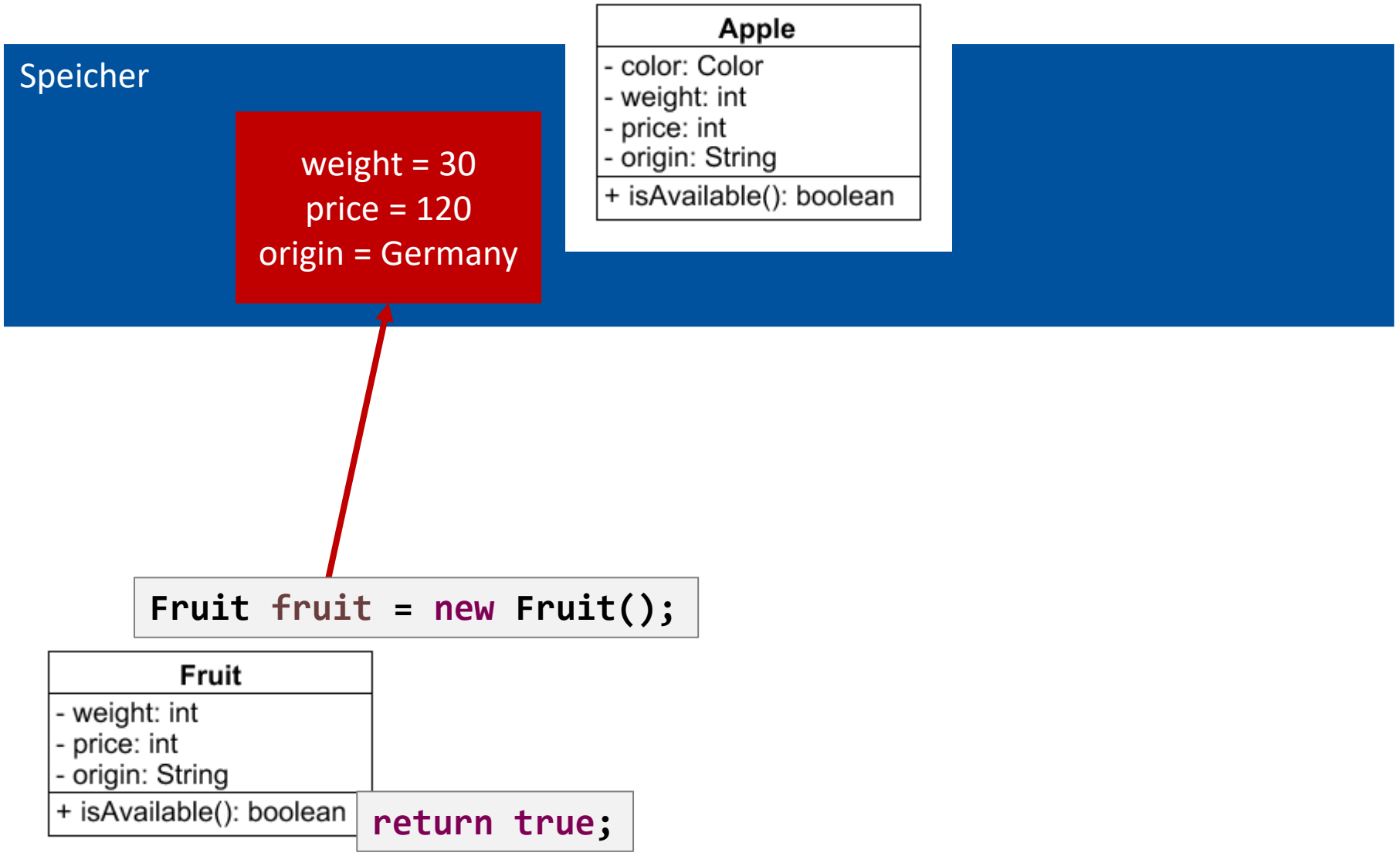
Überschreiben von Methoden (2)



Überschreiben von Methoden (2)



Überschreiben von Methoden (2)



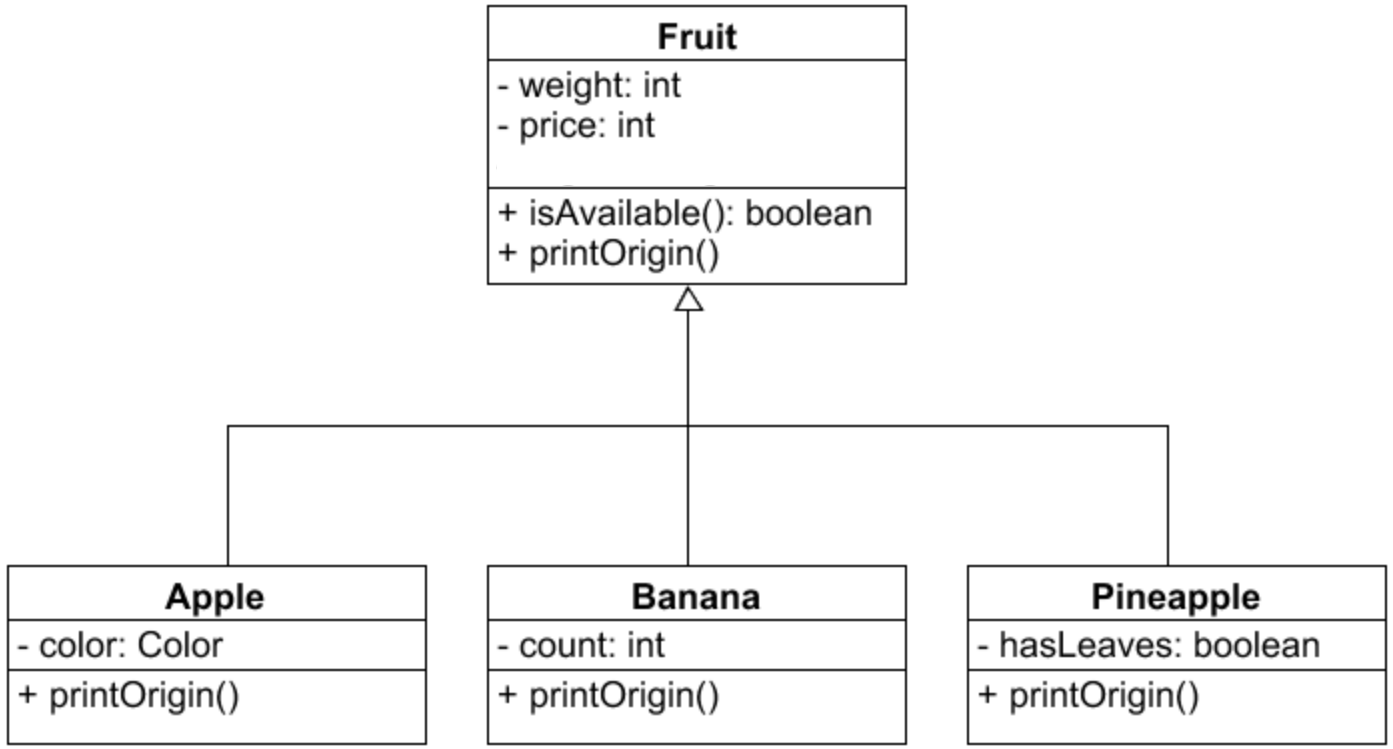
CARL VON OSSIETZKY universität OLDENBURG

Überschreiben von Methoden (3)

```
public class Apple extends Fruit {

    private Color color;

    @Override
    public boolean isAvailable() {
        if (color == Color.RED) {
            return super.isAvailable();
        }
        return false;
    }

}
```

Zugriff auf die Implementierung der Oberklasse über das Schlüsselwort **super**.

Beispiel: Überschreiben und Polymorphie



CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Überschreiben und Polymorphie

```java
public class Fruit {
    public void printOrigin() {
        System.out.println("Country of origin: ?");
    }
}
```

```java
public class Apple extends Fruit {
    @Override
    public void printOrigin() {
        System.out.println("Country of origin: Germany");
    }
}
```

```java
public class Banana extends Fruit {
    @Override
    public void printOrigin() {
        System.out.println("Country of origin: India");
    }
}
```

```java
public class Pineapple extends Fruit {
    @Override
    public void printOrigin() {
        System.out.println("Country of origin: Brazil");
    }
}
```

Beispiel: Überschreiben und Polymorphie

```
public class FruitBasket3 {

    public static void main(String[] args) {
        Fruit[] basket = new Fruit[3];
        basket[0] = new Apple();
        basket[1] = new Banana();
        basket[2] = new Pineapple();
        for (Fruit fruit : basket) {
            fruit.printOrigin();
        }
    }
}
```

```
Country of origin: Germany
Country of origin: India
Country of origin: Brasil
```

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Dynamisches Binden

```
public class FruitBasket3 {
    public static void main(String[] args) {
        Fruit[] basket = new Fruit[3];
        basket[0] = createFruit();
        basket[1] = createFruit();
        basket[2] = createFruit();
        for (Fruit fruit : basket) {
            fruit.printOrigin();
        }
    }
    private static Fruit createFruit() {
        int number = new Random().nextInt(3);
        if (number == 0) {
            return new Apple();
        } else if (number == 1) {
            return new Banana();
        } else {
            return new Pineapple();
        }
    }
}
```

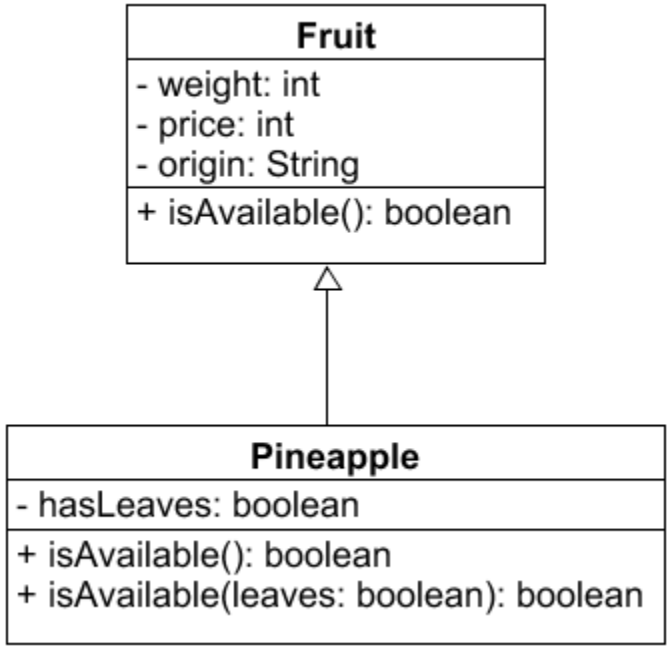
```
Country of origin: ???
Country of origin: ???
Country of origin: ???
```

CARL VON OSSIETZKY universität OLDENBURG

Dynamisches Binden

- Welche konkrete Methodenimplementierung ausgeführt wird, bestimmt der Java-Compiler zur Laufzeit abhängig vom tatsächlichen Typ eines Objekts
  - zur Compile-Zeit ist oft nur der Typ der Referenzvariablen bekannt, aber nicht, mit welchem Typ diese Variable instanziiert wird
  - welche konkrete Implementierung an diese Variable gebunden wird (dynamische Bindung) ist oft erst zur Laufzeit bekannt

Überladen, Überschreiben und Vererbung



CARL VON OSSIETZKY universität OLDENBURG

Überladen, Überschreiben und Vererbung

```
public class Pineapple extends Fruit {

    private boolean hasLeaves;

    @Override
    public boolean isAvailable() {
        return super.isAvailable();
    }

    public boolean isAvailable(boolean leaves) {
        return leaves == hasLeaves;
    }

}
```

CARL VON OSSIETZKY universität OLDENBURG

Polymorphie

- Dynamisches Binden
  - Objekte treten in Gestalt verschiedener Klassen auf
- Überladen
  - Methoden treten in ähnlicher Gestalt bei gleichem Namen auf
- Überschreiben
  - Methoden treten in gleicher Gestalt mit anderem Verhalten auf

Vorteile der Polymorphie

- Wiederverwendbarkeit der Oberklassen in unterschiedlichen Situationen
- Anpassbarkeit an neue Anforderungen:
  neue Anforderung → neue Unterklasse erstellen
- Verschiedene Unterklassen auf die gleiche Art ansprechen, aber spezifische Implementierung je Unterklasse

Beispiel ohne Polymorphie

```
for (Fruit fruit : basket) {
    fruit.printOrigin();
}
```

```
for (Apple apple : appleBasket) {
    apple.printOrigin();
}
for (Banana banana : bananaBasket) {
    banana.printOrigin();
}
for (Pineapple pineapple : pineappleBasket) {
    pineapple.printOrigin();
}
```

Nachteil: Der Code muss jedes Mal, wenn eine neue Unterklasse von **`Fruit`** definiert wird, überall angepasst werden, wo solch eine Unterscheidung auftritt.

CARL VON OSSIETZKY universität OLDENBURG

Vererbung

Fortgeschrittene Konzepte

CARL VON OSSIETZKY universität OLDENBURG

Vererbung und Klassenmethoden

- Klassenmethoden (**`static`**) können **nicht** überschrieben (override) werden.
- Eine Klassenmethode in einer Unterklasse, welche die gleiche Signatur hat wie eine Klassenmethode in der Oberklasse, versteckt (hide) die andere Klassenmethode
- Welche Methode verwendet wird, hängt davon ab, über welchen Typ die Klassenmethode aufgerufen wird

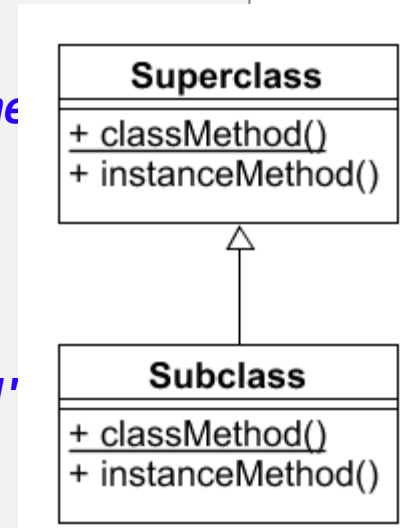
CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Klassenmethoden

```
class Superclass {
    public static void classMethod() {
        System.out.println("Superclass: class method");
    }
    public void instanceMethod() {
        System.out.println("Superclass: instance method");
    }
}
class Subclass extends Superclass {
    public static void classMethod() {
        System.out.println("Subclass: class method");
    }
    @Override
    public void instanceMethod() {
        System.out.println("Subclass: instance method");
    }
}
```

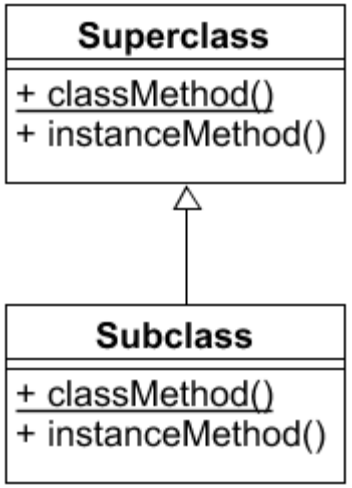
Beispiel: Klassenmethoden

```
class Superclass {
    public static void classMethod() {
        System.out.println("Superclass: class method");
    }
    public void instanceMethod() {
        System.out.println("Superclass: instance me
    }
}
class Subclass extends Superclass {
    public static void classMethod() {
        System.out.println("Subclass: class method"
    }
    @Override
    public void instanceMethod() {
        System.out.println("Subclass: instance method");
    }
}
```



Beispiel: Klassenmethoden

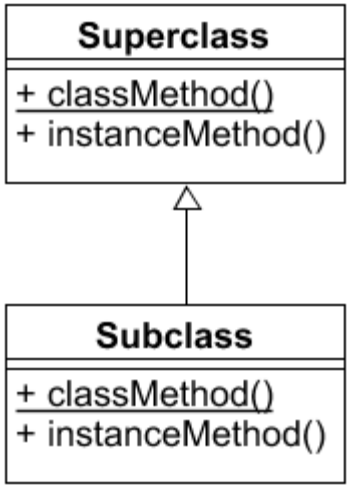
```
public class StaticInheritance {

    public static void main(String[] args) {
        Superclass.classMethod();
        // Superclass: class method

        Superclass superVar = new Subclass();
        superVar.instanceMethod();
        // Subclass: instance method

        Subclass.classMethod();
        // Subclass: class method

        Subclass subVar = new Subclass();
        subVar.instanceMethod();
        // Subclass: instance method

        …
```



Beispiel: Klassenmethoden

```
        …

        superVar.classMethod();
        // Superclass: class method

        subVar.classMethod();
        // Subclass: class method

    }

}
```

The static method **`classMethod()`** from the type **`Superclass/Subclass`** should be accessed in a static way



Vererbung und Klassenattribute

- Klassenattribute (**`static`**) werden in Java zwischen der Ober- und der Unterklasse geteilt

```
public class Fruit {
    protected static int counter = 0;
    public static void printCounter() {
        System.out.println(counter);
    }
}
```

```
public class Apple extends Fruit {
    public Apple() {
        counter++;
    }
}
```

```
Apple apple = new Apple();
Fruit.printCounter(); // 1
```

Erinnerung: `protected`

- Sichtbarkeitsmodifikator
- Zugriff nur von der definierenden Klasse, von allen Unterklassen und vom gleichen Package
- Symbol in der UML: #

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: **protected**

```
public class Fruit {
    private String origin;
}
```

```
public class Apple extends Fruit {

    @Override
    public boolean isAvailable() {
        if (origin.equals("Germany")) {
            return super.isAvailable();
        }
        return false;
    }

}
```

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: **protected**

```
public class Fruit {
    private String origin;
}
```

```
public class Apple extends Fruit {

    @Override
    public boolean isAvailable() {
        if (origin.equals("Germany")) {
            return true;
        }
        return false;
    }

}
```

The Field **Fruit.origin** is not visible

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: **protected**

```
public class Fruit {
    private String origin;
}
```

```
public class Apple extends Fruit {

    @Override
    public boolean isAvailable() {
        if (getOrigin().equals("Germany")) {
            return super.isAvailable();
        }
        return false;
    }

}
```

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: **protected**

```
public class Fruit {
    protected String origin;
}
```

```
public class Apple extends Fruit {

    @Override
    public boolean isAvailable() {
        if (origin.equals("Germany")) {
            return super.isAvailable();
        }
        return false;
    }

}
```

Beispiel: Verdecken von Attributen

```java
public class Fruit {
    private String origin;
}
```

```java
public class Apple extends Fruit {
    private String origin;
}
```

Ein zusätzliches Attribut mit gleichem Namen in der Unterklasse
verdeckt nur das Attribut in der Oberklasse.
→ mit welchem konkreten Attribut (**`Fruit.origin`** oder **`Apple.origin`**)
gearbeitet wird, hängt vom Kontext ab!
→ verwirrend und fehleranfällig
→ verboten!

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Verdecken von Attributen (2)

```
public class Fruit {
    private String origin = "France";
    public void printOrigin() {
        System.out.println("Country of origin: " + origin);
    }
}
```

```
public class Apple extends Fruit {
    private String origin = "Germany";
    @Override
    public void printOrigin() {
        super.printOrigin();
    }
}
```

```
public class FruitBasket {
    public static void main(String[] args) {
        Apple apple = new Apple();
        apple.printOrigin();
        // Country of origin: France
    }
}
```

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Verdecken von Attributen (3)

```
public class Fruit {
    private String origin = "France";
    public void printOrigin() {
        System.out.println("Country of origin: " + origin);
    }
}
```

```
public class Apple extends Fruit {
    private String origin = "Germany";
    @Override
    public void printOrigin() {
        System.out.println("Country of origin: " + origin);
    }
}
```

```
public class FruitBasket {
    public static void main(String[] args) {
        Apple apple = new Apple();
        apple.printOrigin();
        // Country of origin: Germany
    }
}
```

Beispiel: Verdecken von Attributen (4)

```
public class Fruit {
    private String origin = "France";
    public void printOrigin() {
        System.out.println("Country of origin: " + origin);
    }
}
```

```
public class Apple extends Fruit {
    private String origin = "Germany";
}
```

```
public class FruitBasket {
    public static void main(String[] args) {
        Apple apple = new Apple();
        apple.printOrigin();
        // Country of origin: France
    }
}
```

CARL VON OSSIETZKY universität OLDENBURG

Lernziele

- Überladen
- Überschreiben
- Dynamisches Binden
- Vererbung