

# Programmierungsmethodik 1

## Programmierertechnik

**Assoziationen, Basisklasse  
Object, Rekursion**

# Wiederholung

- Abstrakte Basisklassen

# Ausblick



# Worum gehts?



# Agenda

- Beziehungen zwischen Klassen
- Basisklasse Object
- Rekursion



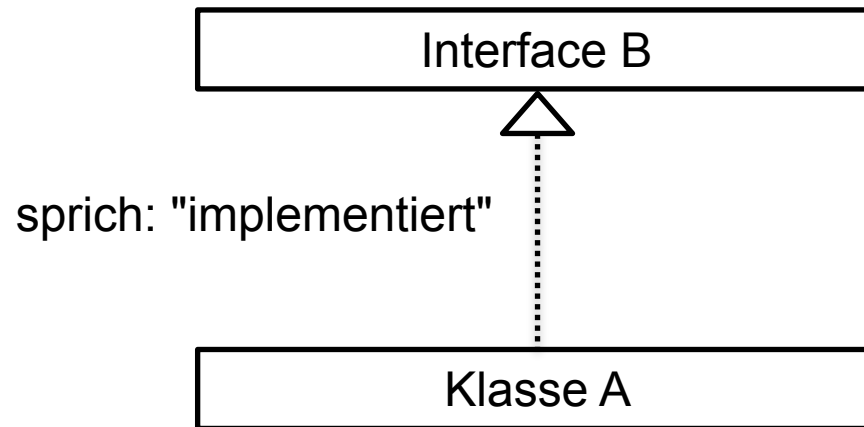
# Beziehungen zwischen Klassen

# Einführung

- UML-Klassendiagramm
  - bisher: Eigenschaften einer Klasse (Name, Objektvariablen, Methoden)
  - außerdem möglich: Beziehungen zwischen Klassen
- Veranschaulichung der Abhängigkeiten zwischen Klassen

# Implementierung einer Schnittstelle

- Java: implements

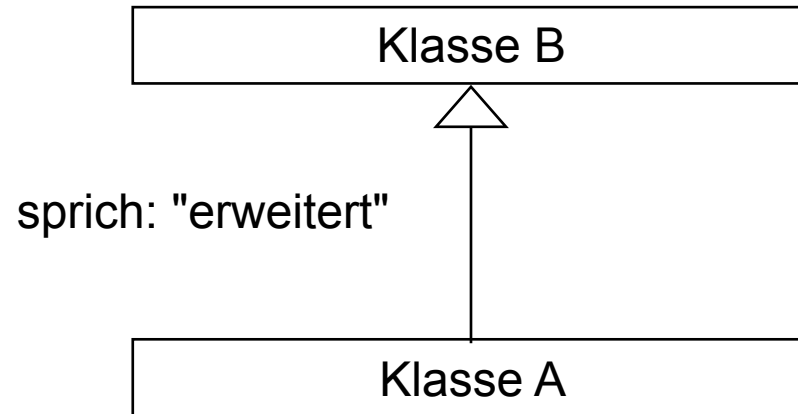


- Beispiel:
  - Klasse Aktiendepot implementiert das Interface Vermoegenswert



# Generalisierung, Vererbung

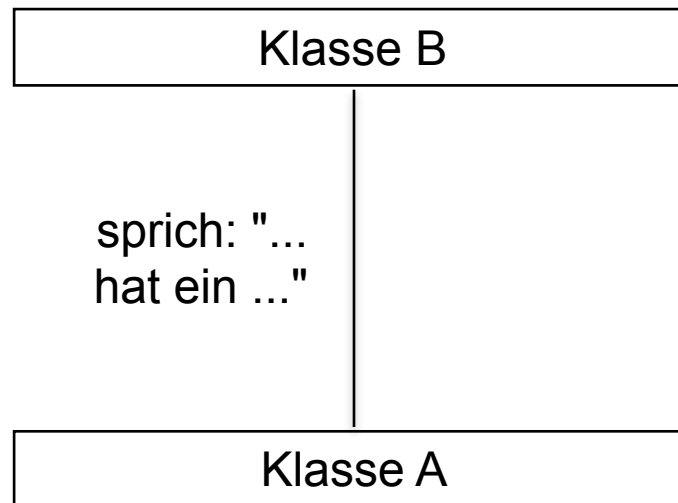
- Java: extends



- Beispiel:
  - Klasse SpeicherZaehler erbt von der Klasse Zaehler
  - oder
  - Klasse SpeicherZaehler erweitert die Klasse Zaehler

# Assoziation

- Java: A hat eine Objektvariable vom Typ B
- auch genannt: Abhängigkeit, Aggregation, Komposition



- Beispiel:
  - Klasse Fahrzeug hat eine Objektvariable vom Typ Lenkrad

# Spezialisierung

- Kriterien für eine Spezialisierung
  - Substitutionsregel: ein Objekt der spezielleren (abgeleiteten) Klasse kann jederzeit anstelle eines Objekts der generelleren (Basis-)Klasse stehen.
  - Umkehrung gilt nicht
- Spezialisierungs-Prinzip
  - Spezialisierung verändert das Verhalten (Methoden) bzw. führt neue Eigenschaften (Objektvariablen) ein
  - Folgerung: wenn sich lediglich die Werte von Eigenschaften (Objektvariablen) ändern, handelt es sich nicht um eine speziellere Klasse, sondern um eine Instanz (Objekt).

# Daumenregel/Best Practice

- Komposition statt Vererbung
- Lässt sich ein Problem sowohl durch Komposition ("hat ein") als auch durch Vererbung lösen, ist Komposition vorzuziehen.

# Übung: Beziehungen zwischen Klassen

- In einer historischen Handelssimulation gibt es folgende Entitäten (Interfaces und Klassen). Erstellen Sie ein Klassendiagramm:
  - Fahrzeug
  - Schiff
  - Segel
  - Kaufmann(-frau)
  - Pferdegespann
  - Pferd
  - Wagen
  - Kogge





# Basisklasse Object

# Class-Objekt

- zu jedem Typ (Klasse, Interface, primitiver Typ) existiert genau ein Typobjekt (repräsentiert den Typ)
- Typobjekte sind technisch Instanzen der Klasse Class
  - mit eigenen Methoden wie z.B. getName()
- Zugriff auf das Typobjekt eines Objekts:  
`<Objekt>.getClass()`
- Alternative Beispiel-Lösung ohne instanceof-Operator
  - viel weniger schöne Lösung!

```
if (zaehler.getClass().getName().equals("SpeicherZaehler")) {  
    // nur für SpeicherZaehler  
    ((SpeicherZaehler) zaehler).speichern();  
}
```

# Class-Objekt

- Class-Objekt bietet noch viel mehr Möglichkeiten
- Abfrage von Eigenschaften einer Klasse zur Laufzeit
- zur Vertiefung → Reflection



# Basisklasse aller anderen Klassen: Object

- Object ist voreingestellte Basisklasse aller Klassen
  - Teil der Java-Laufzeitbibliothek im Package java.lang
  - "Wurzel" des Ableitungsbaums
  - jede Klasse ist abgeleitet, außer Object
  - Alle Klassen (außer Object) haben, direkt oder indirekt, Object als gemeinsame Basisklasse
- äquivalent:

```
public class <Klassenname> {...}
```
- und

```
public class <Klassenname> extends Object {...}
```
- Methoden von Object werden an jede Klasse vererbt

# Vordefinierte Methoden in Object

- Object-Methoden bieten zum Teil nur minimale Funktionalität und sollten vor Gebrauch redefiniert werden (Vererbung!):
  - toString() liefert classname@hashcode
  - equals() prüft Identität (wie der Vergleich mit ==), nicht inhaltliche Gleichheit
  - hashCode() verwendet die Speicheradresse für eine Kennnummer

<b>public String toString()</b>	<b>lesbare Repräsentation</b>
public boolean equals(Object obj)	true wenn dieses Objekt und obj identisch sind, false ansonsten
public int hashCode()	Kennnummer
protected Object clone()	Erzeugt ein Duplikat
public Class getClass()	Liefert das Typobjekt

# toString()

- Beschreibung der aktuellen Instanz (meist des aktuellen Zustands)
  - z.B. Name der Klasse
  - Belegung der Objektvariablen
- Beispiel aus Bruch:

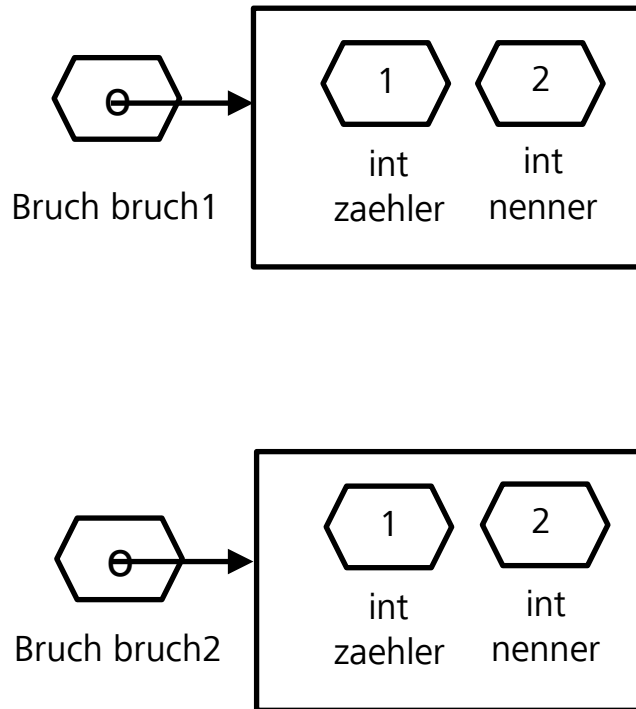
```
@Override  
public String toString() {  
    return String.format("%d/%d", zaehler, nenner);  
}
```

# Inhaltlicher Vergleich mit equals

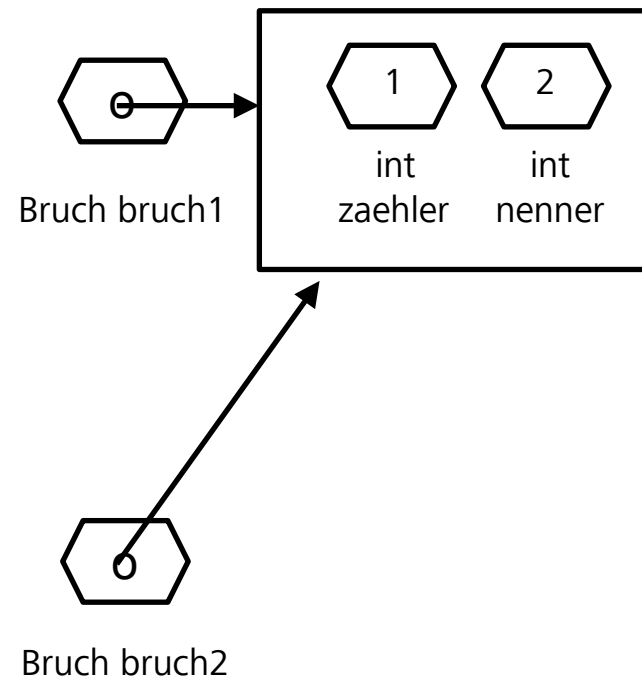
- Idee: zwei Objekte sind für equals() gleich, wenn sie für einen Anwender ausgetauscht werden könnten (gleiche Inhalte)
- Die equals()-Implementierung der Klasse Object prüft aber nur Objekt-Identität statt Gleichheit

```
Bruch bruch1 = new Bruch(1, 2);  
Bruch bruch2 = new Bruch(1, 2);  
System.out.println(bruch1.equals(bruch2)); // false
```

```
Bruch bruch1 = new Bruch(1, 2);  
Bruch bruch2 = new Bruch(1, 2);
```



```
Bruch bruch1 = new Bruch(1, 2);  
Bruch bruch2 = bruch1;
```



# Redefinition vs. Überladen

- Signatur von equals in der Klasse Object:

```
public boolean equals(Object obj)
```

Argument: Objekt beliebigen Typs

- Ziel: Redefinition von equals() für inhaltlichen Vergleich!
- populärer Fehler: Definition von equals() mit anderem Parametertyp als Object, beispielsweise

```
public boolean equals(Bruch bruch) // statt Object obj
```

- dann aber: Überladen statt Redefinition
  - neue Methodensignatur!


# Inhaltlicher Vergleich mit equals

- Beispiel: equals()-Implementierung für die Klasse Bruch


@Override

```
public boolean equals(Object anderesObject) {  
    if (!(anderesObject instanceof Bruch)) {  
        return false;  
    }  
    Bruch andererBruch = (Bruch) anderesObject;  
    return (zaehler == andererBruch.zaehler) &&  
        (nenner == andererBruch.nenner);  
}
```

Schritt1: Prüfen, ob das  
andere Objekt kompatibel  
ist



oftmals: paarweiser Vergleich  
aller Objektvariablen



# Vergleich der Objektvariablen

- im letzten Schritt paarweiser Vergleich aller Objektvariablen
  - nicht vergessen: ererbte Objektvariablen!
  - primitive Typen (nicht double, float): Vergleich mit == oder !=
  - Referenztypen: Vergleich mit equals()
- Typcast gefahrlos möglich wegen vorausgegangener Typprüfung
- equals() funktioniert nur dann, wenn alle beteiligten Klassen ebenfalls eine korrekte Implementierung definieren!



# Übung: Equals

- Überschreiben Sie in den beiden Klassen jeweils die equals()-Methode.

```
public class Person {  
    private String name;  
  
    public Person(String name) {  
        this.name = name;  
    }  
}
```

```
public class Konto {  
    private double kontostand;  
  
    private Person kontoinhaber;  
  
    public Konto(double  
        kontostand, Person  
        kontoinhaber) {  
        this.kontostand =  
            kontostand;  
        this.kontoinhaber =  
            kontoinhaber;  
    }  
}
```

# HashCode

- für einzelnes Objekt charakteristischer int-Wert
- viele Methoden der Laufzeitbibliothek benutzen equals() in Kombination mit hashCode() zur Effizienzsteigerung
- Methode hashCode liefert einen Hashcode eines Objektes aufgrund der internen Darstellung im Speicher, Beispiele:

`"Hello".hashCode() → 69609650`

`"World".hashCode() → 83766130`

# HashCode

- konkreter Zahlenwert irrelevant, aber zwei Bedingungen:
  - Konsistenz: gleiche Objekte müssen gleiche Hashcodes haben (`x.hashCode() == y.hashCode()`)
  - Effizienz: verschiedene Objekte sollten möglichst verschiedene Hashcodes haben (`x.hashCode() != y.hashCode()`)
- wenn `equals()` redefiniert wurde, sollte `hashCode()` ebenfalls redefiniert werden
- Ergebnisberechnung von `hashCode()` aufgrund der aktuellen Objektvariablen!

# Equals und hashCode

- Klasse BeschraenkterZaehler

```
public int hashCode() {  
    return getWert() * getGrenze();  
}
```

- in equals() verwenden vor paarweisem Vergleich aller Objektvariablen:

```
if( anderesObjekt == null || // ungleich null  
    getClass() != other.getClass() || // gleiche Klasse  
    hashCode() != other.hashCode() ){ // Inhalt ggf. gleich  
    // dann: paarweiser Vergleich der Objektvariablen  
    ...  
}
```

# Übung: hashCode

- Überschreiben Sie die Methode hashCode() in der folgenden Klasse. Der Wert für zwei Instanzen der Klasse soll nur gleich sein, wenn beide Instanzen die gleichen Werte für ihre Objektvariablen haben.

```
public class WahrheitUndBuchstabe {  
    private boolean wahrheitswert;  
  
    private char buchstabe;  
  
    public WahrheitUndBuchstabe(boolean wahrheitswert, char  
        buchstabe) {  
        this.wahrheitswert = wahrheitswert;  
        this.buchstabe = buchstabe;  
    }  
}
```



# Rekursion

# Rekursion

- Oft: Problem lässt sich auf kleinere Version des gleichen Problems reduzieren
- Beispiel: Berechnung der Fakultät
$$\text{fak}(4) = 4 * 3 * 2 * 1 = 4 * \text{fak}(3)$$
  - irgendwann: triviales Problem, Lösung bekannt
$$\text{fak}(1) = 1$$
  - allgemein:  $\text{fak}(n) = n * \text{fak}(n-1)$
- Zusammenfassung:
  - Lösung eines Problems durch Lösung einer einfacheren Version des Problems
  - Ende: triviale Abbruchbedingung

# Rekursion in Java

- Lösung eines Problems mit einer Methode
- Aufruf derselben Methode aus dem eigenen Methodenrumpf heraus
  - "rekursiver Methodenaufruf"
  - "Selbstbezüglichkeit"
- vor rekursivem Aufruf: Test auf Abbruchbedingung



# Beispiel: Fakultät

```
public static int fakultaet(int zahl) {  
    if (zahl == 1) {  
        return 1;  
    }  
    return zahl * fakultaet(zahl - 1);  
}
```

# Abbruchbedingung

- Abbruchbedingung spielt zentrale Rolle
- es muss sichergestellt sein, dass diese immer erreicht wird
- ansonsten: endlose Laufzeit
- hier (Fakultät):
  - Forderung: Eingabe muss  $\geq 1$  sein
  - rekursiver Aufruf mit um 1 kleinerer Zahl
  - daher: Abbruchbedingung Zahl 1 muss erreicht werden

# Rekursion vs. Iteration

- jede rekursive Formulierung kann auch durch iterative Formulierung (Schleife) ersetzt werden
  - und umgekehrt
- Beispiel: iterative Berechnung der Fakultät

```
public static int fakultaetIterativ(int zahl) {  
    int ergebnis = 1;  
    for (int zaehler = 2; zaehler <= zahl; zaehler++) {  
        ergebnis *= zaehler;  
    }  
    return ergebnis;  
}
```

## Übung: Summe

- Schreiben Sie eine rekursive Methode zur Berechnung der Summe der ganzen Zahlen von  $1 \dots n$ .

# Übung: Rekursion statt Schleife

- Schreiben Sie eine rekursive Methode `erzeugeAlphabetRekursiv()`, die das gleiche Ergebnis wie die folgende iterative Variante erzeugt.

```
public static String erzeugeAlphabetIterativ() {  
    String ergebnis = "";  
    for (int i = 0; i < 26; i++) {  
        ergebnis += (char) ('a' + i);  
    }  
    return ergebnis;  
}
```

# Vorteile

- oft kompakte Formulierung einer Lösung
- oft gut lesbare Formulierung einer Lösung
- viele Lösungen sind an sich "rekursiv"
  - z.B. Durchlaufen von Baumstrukturen

# Nachteile

- ja nach Programmiersprache Speicherprobleme bei hoher Rekursionstiefe
- je nach Geschmack schlechter lesbar
- teilweise Hilfsmethoden notwendig (Parameter wie z.B. Zähler)

## Übung: Addition

- Implementieren Sie einen rekursiven Algorithmus, der die Summe  $a + b$  zweier natürlicher Zahlen rekursiv berechnet. Dabei sind als arithmetische Funktion lediglich das Addieren von 1 zu einer Zahl oder das Subtrahieren von 1 von einer Zahl erlaubt.



# Zusammenfassung

- Beziehungen zwischen Klassen
- Basisklasse Object
- Rekursion