



Programmierungsmethodik 1

Programmiertechnik

Assoziationen, Basisklasse
Object, Rekursion

- Vererbung
- Methoden
- Konstruktor und Objektvariablen
- Abstrakte Basisklassen

Ausblick für heute

- Bei einer gemeinsamen Software-Architektur sollen verschiedene Beziehungen zwischen den Komponenten eines Programms (Klassen, Interfaces) dargestellt werden.
- Was mache ich, wenn ich eine Variable brauche, die auf ein beliebiges Java-Objekt zeigen können soll?
- Wie gehe ich damit um, dass eine Lösungsroutine für ein Problem, sich scheinbar selber wieder aufrufen muss (z.B. mit einem veränderten Argument)?

Agenda

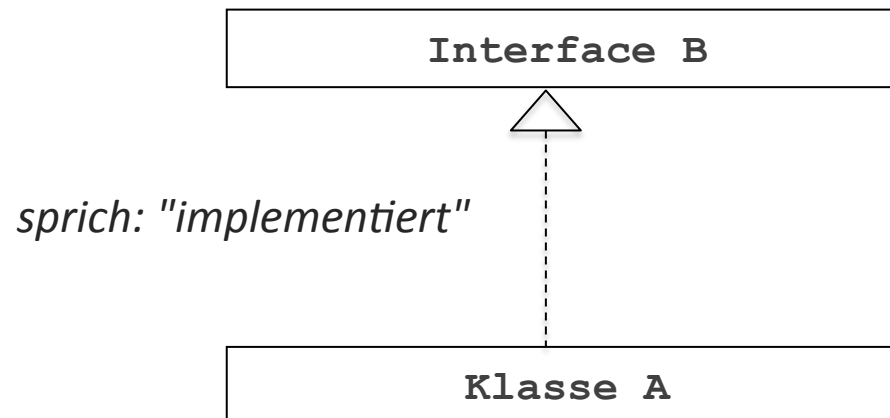


- Beziehungen zwischen Klassen
- Basisklasse Object
- Rekursion

Beziehungen zwischen Klassen

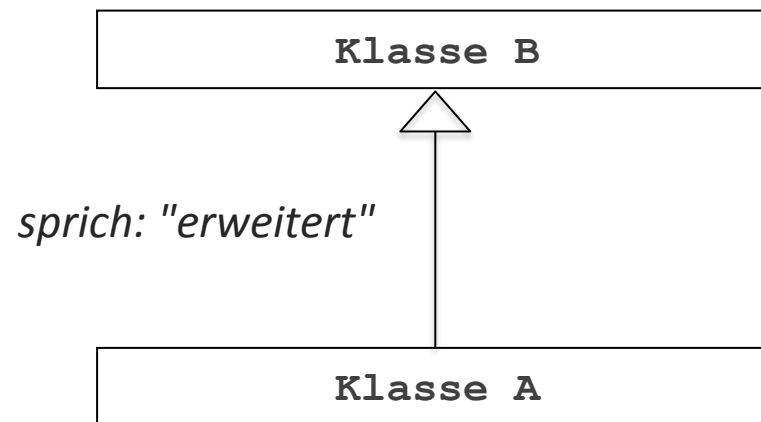
- UML-Klassendiagramm
 - bisher: Eigenschaften einer Klasse (Name, Objektvariablen, Methoden)
 - außerdem möglich: Beziehungen zwischen Klassen
- Veranschaulichung der Abhängigkeiten zwischen Klassen

- Java: implements



- Beispiel:
 - Klasse Aktiendepot implementiert das Interface Vermoegenswert

- Java: extends



- Beispiel:
 - Klasse SpeicherZähler erbt von der Klasse Zähler
 - oder
 - Klasse SpeicherZähler erweitert die Klasse Zähler

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

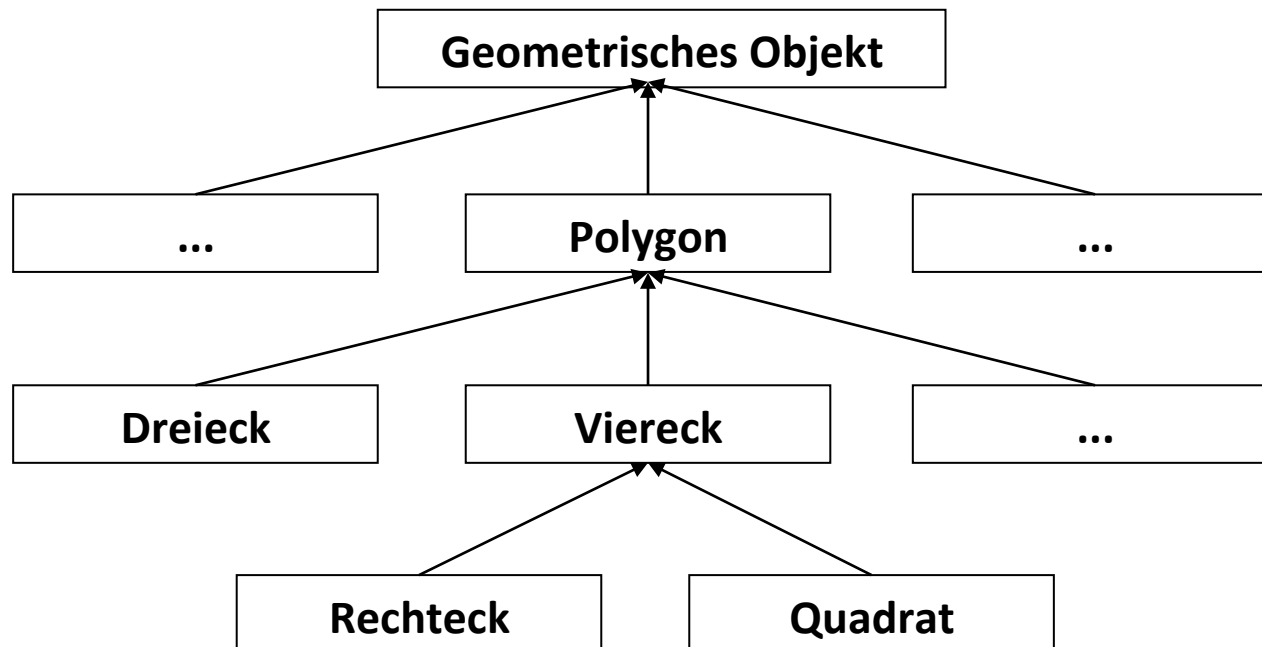


- Beispiel:
 - Klasse Fahrzeug hat eine Objektvariable vom Typ Lenkrad

- 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).

Beispiel Substitutionsregel

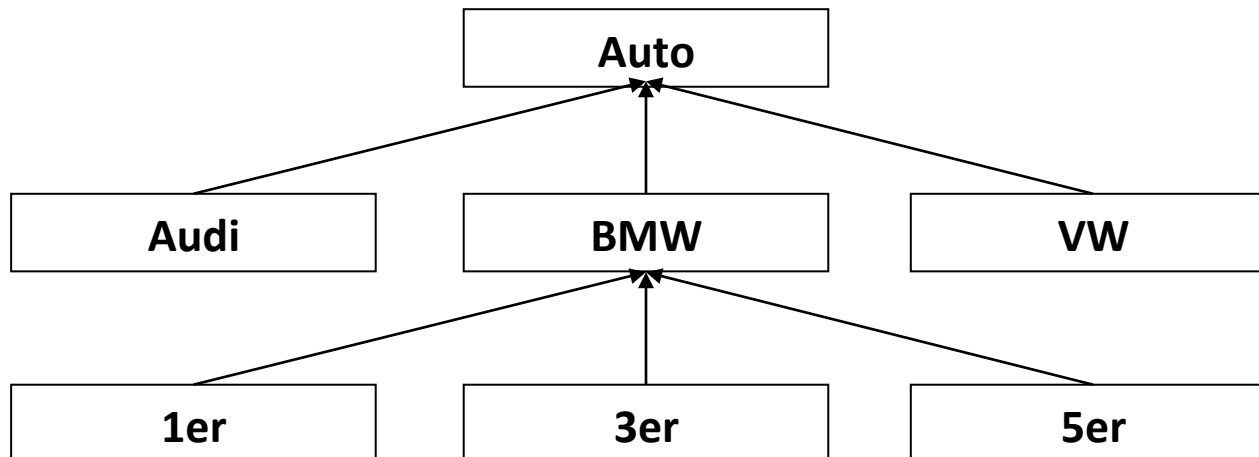
- Mathematik: Jedes Quadrat ist ein Rechteck → Spezialisierung
- Problem: für ein Rechteck könnten Methoden definiert werden, die für ein Quadrat ungültig sind (z.B. `halbiereBreite()`)



Beispiel Spezialisierungs-Prinzip

- dies ist keine sinnvolle Spezialisierung, solange keine firmenspezifischen Methoden/Variablen benötigt werden!
- ansonsten sind dies "normale" Objekte der Klasse `Auto`:

```
public String hersteller = "BMW";  
public String modell = "1er";
```



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

- 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

- 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")) {  
    ((SpeicherZaehler) zaehler).speichern(); // nur für SpeicherZaehler  
}
```

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

- 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

- 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

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

- 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);  
}
```

- 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
```

- 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!

- 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);
}
```

Schritt 1: Prüfen, ob
das andere Objekt
kompatibel ist



oftmals: paarweiser
Vergleich aller
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!

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

```
/**
 * Eine sehr einfache Repräsentation für ein Konto bestehend aus
 * einem
 * Kontostand und einem Kontoinhaber.
 */
public class Konto {

    /**
     * Aktueller Kontostand
     */
    private double kontostand;

    /**
     * Inhaber des Kontos.
     */
    private Person kontoinhaber;

    /**
     * Konstruktor.
     */
    public Konto(double kontostand, Person kontoinhaber) {
        this.kontostand = kontostand;
        this.kontoinhaber = kontoinhaber;
    }
}
```

```
/**
 * Einfache Repräsentation für eine Person.
 */
public class Person {

    /**
     * Name der Person.
     */
    private String name;

    /**
     * Konstruktor.
     */
    public Person(String name) {
        this.name = name;
    }
}
```

- 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

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

- 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 könnte gleich sein  
    // dann: paarweiser Vergleich der Objektvariablen  
    ...  
}
```

- 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.

```
/**
 * Diese Klasse repräsentiert die Kombination aus einem Wahrheitswert und einem
 * Buchstaben (a-z).
 */
public class WahrheitUndBuchstabe {

    /**
     * Aktueller Wahrheitswert.
     */
    private boolean wahrheitswert;

    /**
     * Aktueller Buchstabe, es sind nur Kleinbuchstaben a-z erlaubt.
     */
    private char buchstabe;

    /**
     * Konstruktor.
     */
    public WahrheitUndBuchstabe(boolean wahrheitswert, char buchstabe) {
        this.wahrheitswert = wahrheitswert;
        this.buchstabe = buchstabe;
    }
}
```

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

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

```
/**  
 * Berechnet die Fakultät einer ganzen Zahl größer 0 (rekursive  
 * Variante).  
 */  
public static int fakulaet(int zahl) {  
    if (zahl == 1) {  
        return 1;  
    }  
    return zahl * fakulaet(zahl - 1);  
}
```

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

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

```
/**  
 * Berechnet die Fakultät einer ganzen Zahl größer 0 (iterative  
 Variante).  
 */  
public static int fakultaetIterativ(int zahl) {  
    int ergebnis = 1;  
    for (int zaehler = 2; zaehler <= zahl; zaehler++) {  
        ergebnis *= zaehler;  
    }  
    return ergebnis;  
}
```

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

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

```
/**
 * Iterative Erzeugung einer Alphabet-Zeichenkette
 * ("abcdefghijklmnopqrstuvwxyz").
 */
public static String erzeugeAlphabetIterativ() {
    String ergebnis = "";
    for (int i = 0; i < 26; i++) {
        ergebnis += (char) ('a' + i);
    }
    return ergebnis;
}
```

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

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

- 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.

- Beziehungen zwischen Klassen
- Basisklasse Object
- Rekursion