# Programmiermethodik 1
# Programmiertechnik

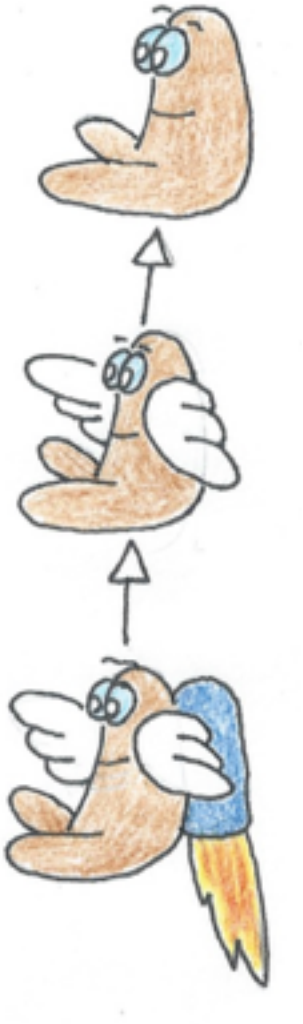
## Vererbung

# Wiederholung

- Interfaces
- Dynamisches Binden
- Arbeiten mit Interfaces

# Ausblick

# Worum gehts?



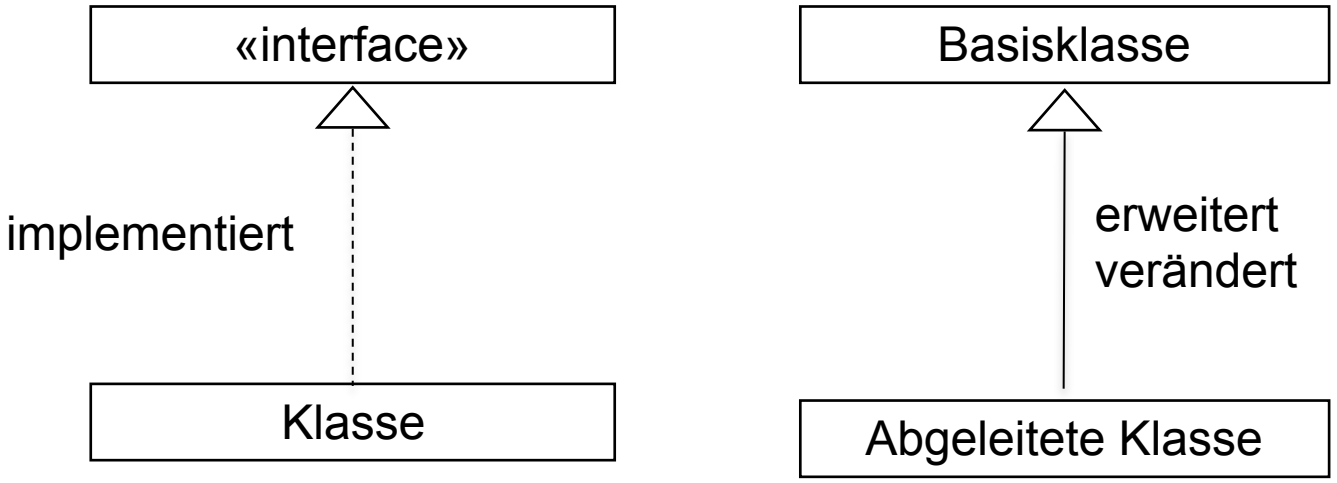
# Agenda

- Vererbung
- Methoden
- Konstruktor und Objektvariablen

# Vererbung: Einführung

# UML: Beziehungen zwischen Klassen und Interfaces

- Interfaces isolieren gleiche Eigenschaften unabhängiger Klassen
- abgeleitete Klassen "erben" die Eigenschaften von Basisklassen und erweitern oder verändern diese Eigenschaften



# Beispiel: Zähler

- Ein Zähler hat einen Zählerstand (ganze, nicht-negative Zahl), dieser kann
  - weitergezählt,
  - abgelesen und
  - auf 0 zurückgestellt
  - werden.

# Beispiel: Zähler

```
public class Zaehler {
   protected int wert = 0;

   public void erhoehen( ) {
      wert++;
   }

   public int getWert( ) {
      return wert;
   }

   public void reset( ) {
      wert = 0;
   }
}
```

```
Zaehler zaehler1 = new
   Zaehler( );
for (int i = 0; i < 10; i++) {
   zaehler1.erhoehen( );
   System.out.format("%d ",
      zaehler1.getWert( ));
}
System.out.println( );
```

- Ausgabe:
  *1 2 3 4 5 6 7 8 9 10*

# Erweitern der Funktionalität

- Annahme
  - neue Art von Zähler gefordert
  - kann zusätzlich einen Zählerstand speichern
  - Klasse Zaehler ist "ausgeliefert" und kann nicht mehr verändert werden

- neue Klasse SpeicherZaehler mit Methoden zum
  - speichern aktuellen Zählerstand

```
speichern()
```

  - zurücksetzen auf zuletzt gespeicherten Stand
  - 0, wenn vorher kein Speichern erfolgte
    - wiederherstellen()

# Erste Lösungsideen

- Code von Zaehler kopieren und erweitern
    - Ergebnis: zwei isolierte Klassen ohne Bezug
    - Problem: mögliche Fehler im Code von Zaehler auch in SpeicherZaehler
        - muss zweimal identisch korrigiert werden
- gemeinsames Interface für Zaehler und SpeicherZaehler
    - jetzt: Verwandtschaft im Code dokumentiert
    - aber: keine Hilfe bei Fehlerkorrektur, nur Zusatzaufwand + Aufwand für Interface
- neues Konzept notwendig!

# Vererbung

- neues Konzept
    - Ableitung (engl. derivation) oder Vererbung (engl. inheritance)
- eine neue Klasse wird an eine vorhandene Klasse gebunden und "erbt" deren Objektvariablen und Methoden
- neue Klasse kann weiteren Code hinzufügen oder den geerbten Code verändern
- Bezeichnungen:
    - vorhandene Klasse = Basisklasse (base class, super class)
    - neue Klasse = abgeleitete Klasse (derived class)
- im Beispiel:
    - Basisklasse: Zaehler
    - abgeleitete Klasse: SpeicherZaehler

# Ableiten konkreter Klassen

- Definition einer abgeleiteten Klasse ist reduziert auf die Unterschiede zur Basisklasse
- Schlüsselwort extends koppelt abgeleitete Klasse und Basisklasse
- Syntax:

```
public class <Name der abgeleiteten Klasse> extends
    <Basisklassenname>{
    …
}
```

- "normale" Klassendefinition der abgeleiteten Klasse für die zusätzlichen/ veränderten Variablen und Methoden
- Ableitung ist asymmetrisch:
    - abgeleitete Klasse kennt ihre Basisklasse (siehe Syntax)
    - Basisklasse weiß nichts von abgeleiteten Klassen

# Klasse SpeicherZaehler

```
public class SpeicherZaehler extends Zaehler {
   private int speicher = 0;

   public void speichern( ) {
      speicher = wert;
   }

   public void wiederherstellen( ) {
      wert = speicher;
   }
}
```

# Vergleich der Klassenbestandteile

| **Klasse** | **Zaehler** | **SpeicherZaehler** |
|---|---|---|
| Konstruktor | automatisch | automatisch |
| Objektvariablen | # wert:int | (ererbt ??) |
| | | - speicher:int |
| Methoden | + erhoehen():void | ererbt |
| | + getWert():int | ererbt |
| | + reset():void | ererbt |
| | | + speichern():void |
| | | + wiederherstellen():void |

# Übung: Hund

- Schreiben Sie ein Interface Tier. Jedes Tier kann ein Geräusch machen.
- Implementieren Sie eine Klasse Hund, der bekanntlich ein Tier ist.
- Schreiben Sie außerdem eine Klasse Rettungshund. Ein Rettungshund kann alles, was ein normaler Hund kann, und außerdem Menschen retten.

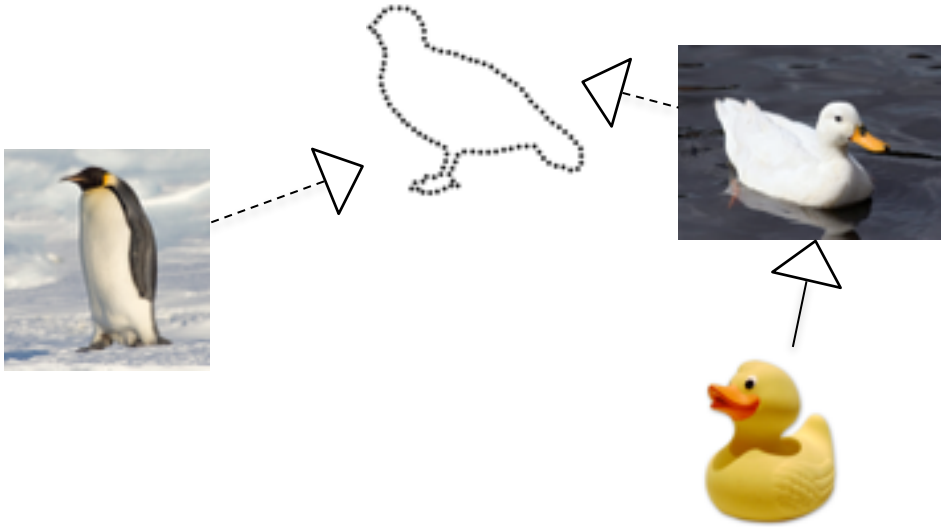
# Vererbung

# Übung: Dynamische Bindung

```java
public interface Vogel {
   public void fliegen();
   public void singen();
}

public class Pinuguin implements Vogel {
   public void fliegen() {
      System.out.println("Ich kann nicht
         fliegen :-(");
   }
   public void singen() {
      System.out.println("tröt, tröt");
   }
}

public class Ente implements Vogel {
   public Ente() {
      System.out.println("Ich bin eine Ente!");
      fliegen();
   }
   public void fliegen() {
      System.out.println("Flap, flap");
   }

   public void singen() {
      System.out.println("Quak, quak");
   }
}

public class QuietscheEnte extends Ente {
   public QuietscheEnte() {
      System.out.println("Ich bin eine Quietscheente!");
   }
   public void fliegen() {
      super.fliegen();
      System.out.println("Oh, I vergaß. Ich kann gar nicht fliegen");
   }
   public void singen() {
      System.out.println("Quitsch");
   }
}
```

Was ist die Ausgabe von ...?

```java
Bird bird1 = new Penguin();
bird1.fly();
bird1.sing();

Bird bird2 = new Duck();
bird2.sing();

Bird bird3 = new RubberDuck();
bird3.sing();
```



# Zugriffsschutz

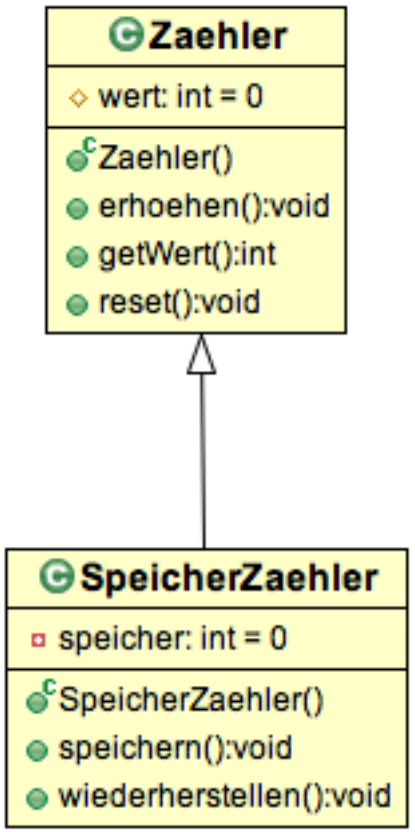
- Objektvariable wert ist  private in Zaehler
    - Zugriff nur in Klasse Zaehler
    - Problem: SpeicherZaehler braucht wert, hat aber keinen Zugriff
        - Compiler verweigert Übersetzung!
    - Lösung: Zugriffsschutz  protected (UML-Abkürzung: #)
- protected-Objektvariablen und -Methoden sind in der Klasse selbst und zusätzlich in allen abgeleiteten Klassen verfügbar
- korrigierte Fassung von Zaehler

```
public class Zaehler {
   protected int wert = 0; // vorher private
    … Rest wie vorher …
}
```

# Zugriffsschutz: UML

- offizielle UML-Syntax für protected: #
- auch verwendet: gelber Diamant

- Textebene 1
  - Te




e 5

# Datenkapselung der Basisklasse

- direkter Zugriff auf Objektvariablen weiterhin fragwürdig
    - ob ererbt oder nicht
- Zugriff über Getter und Setter empfehlenswert!
- Daher besser: zusätzlich Setter in Zaehler definieren

```
public class Zaehler{
private int wert= 0;
...
protected void setWert(int wart) {
   this.wert = wert;
}
```

- in SpeicherZaehler verwenden:

```
setWert(speicher);
speicher = getWert();
```

# Methoden

# Aufruf ererbter Methoden

- für Anwendungen: ererbte Methoden und Methoden einer Klasse selbst sind nicht unterscheidbar
- Methodenaufruf: Die JVM sucht zuerst in der Klasse selbst, dann in der Basisklasse, dann in deren Basisklasse usw.
- Der Compiler stellt sicher, dass die JVM in jedem Fall eine Methode findet

```
SpeicherZaehler speicherZaehler = new SpeicherZaehler ();
speicherZaehler.erhoehen(); // ererbt von Zaehler
speicherZaehler.speichern();
speicherZaehler.reset(); // ererbt von Zaehler
speicherZaehler.wiederherstellen();
```

# Redefinition abgeleiteter Methoden

- eine abgeleitete Klasse kann Methoden redefinieren, die bereits in der Basisklasse definiert sind
    - also neu definieren, überschreiben
- Regeln:
    - Name und Parameterliste müssen exakt übernommen werden
    - Zugriffsschutz darf gelockert werden, aber nicht eingeschränkt
    - Ergebnistyp darf eine entsprechend abgeleitete Klasse sein
    - Rumpf kann komplett ersetzt werden
- Funktionalität der Basisklasse wird hier nicht erweitert, sondern verändert
    - anders als im Beispiel SpeicherZaehler

# Beispiel: Zähler mit Anschlag

- neue Variante von Zählern, die nur bis zu einem bestimmten Grenzwert laufen und dort stehen bleiben
- neue Klasse BeschraenkterZaehler
- BeschraenkterZaehler erbt von ebenfalls von Zaehler
- zusätzlich:
    - final-Objektvariable grenze zum Speichern des Grenzwerts
    - Getter für den Grenzwert
    - Konstruktor zum Initialisieren des Grenzwerts

# Beispiel: Zähler mit Anschlag

```
public class BeschraenkterZaehler extends Zaehler {
  private final int grenze;
  public BeschraenkterZaehler(int grenze) {
    this.grenze = grenze;
  }
  public int getGrenze( ) {
    return grenze;
  }

  @Override
  public void erhoehen( ) { // gleiche Signatur
    if (getWert( ) < grenze) { // neuer Rumpf
      setWert(getWert( ) + 1);
    }
  }

  @Override
  public void setWert(int wert) {
    if (wert > grenze) {
      super.setWert(grenze);
    }
    this.wert = wert;
  }

  public void erhoehen(int schrittweite) {
    wert += schrittweite;
  }
}
```

# Redefinition einer Methode

- BeschraenkterZaehler erbt die Methoden erhoehen(), setWert(), getWert(), reset() von Zaehler
- Methoden sind unverändert brauchbar, außer erhoehen():
    - nicht endlos weiterzählen, sondern am Grenzwert stoppen!
- BeschraenkterZaehler redefiniert die Methode erhoehen():

```
@Override
public void erhoehen() { // gleiche Signatur
   if (getWert() < grenze) { // neuer Rumpf
      setWert(getWert() + 1);
   }
}
```

# Ableiten konkreter Klassen

| Klasse | Zaehler | BeschraenkterZahler |
|---|---|---|
| Konstruktor | automatisch | BeschraenkterZaehler(int) |
| Objekt-variablen | # wert:int | ererbt<br>- grenze:int |
| Methoden | + erhoehen():void | + erhoehen ():void |
| | + setWert(int):void | ererbt |
| | + getWert():int | ererbt |
| | + reset():void | ererbt |
| | | + getGrenze():int |

# Überladen vs. Redefinition

- Vorsicht: bei gleichem Namen und abweichender Parameterliste wird eine ererbte Methode überladen, nicht redefiniert!
- Beispiel:

```
public void erhoehen(int schrittweite) {
   wert += schrittweite;
}
```

- in der Klasse BeschraenkterZahler gibt es nun zwei Methoden erhoehen()
    - die eine ererbt, die andere neu definiert

# Einschränken einer Basisklasse

- abgeleitete Klassen können die Funktionalität Basisklasse erweitern oder ändern, aber keinesfalls einschränken
- kein Sprachmittel zum Ausblenden ererbter Methoden oder Objektvariablen vorhanden
- Beispiel: Redefinition mit reduziertem Zugriffsschutz unzulässig:

```
public class BeschraenkterZaehler extends Zaehler {
   ...
   private void erhoehen(){ ... } // Fehler!
}
```

- Fazit:
  - ein abgeleitetes Objekt bietet als Schnittstelle alles an, was ein Basisklassenobjekt kann
  - möglicherweise auch mehr, aber keinesfalls weniger

# Dynamisches Binden redefinierter Methoden

- abgeleitete Klassen sind kompatibel zu Basisklassen
    - vgl. auch Interfaces
- Folge: Objekt einer abgeleiteten Klasse kann ein Basisklassenobjekt in jedem Kontext ersetzen
- redefinierte Methoden werden dynamisch gebunden
- Beispiel:
    - Erzeugen eines Objekts der Klasse BeschraenkterZaehler statt Zaehler in der Beispielanwendung
    - erhoehen()und getWert() werden dynamisch gebunden
    - die Entscheidung für BeschraenkterZaehler kann erst zur Laufzeit getroffen werden

# Beispiel: BeschraenkterZaehler

```
Zaehler zaehler2 = new BeschraenkterZaehler(5);
for (int i = 0; i < 10; i++) {
   zaehler2.erhoehen();
   System.out.format("%d ", zaehler2.getWert());
}
System.out.println();
```

- Ausgabe
    - 1 2 3 4 5 5 5 5 5 5

# Übung: Kaffeemaschine

- Gegeben ist folgende Klasse KaffeeMaschine.
- Schreiben Sie eine Klasse EspressoMachine. Die macht auch Kaffee, aber besseren (gleiche Methode, andere Ausgabe).
- Außerdem macht die EspressoMachine Cappuccino (dazu braucht man Kaffeepulver und Milch). Verwenden Sie den gleichen Methodenbezeichnet

```
public class KaffeeMaschine {
   public void kaffeeMachen(int mengeKaffee) {
      System.out.format("Lecker Kaffee aus %d
         Gramm Kaffeepulver.\n", mengeKaffee);
   }
}
```

# Konstruktoren und Objektvariablen

# Statisches Binden von Methoden

- Java bindet Methoden als Standard dynamisch
- in einigen Fällen wird statisch gebunden
    - der Compiler ordnet Aufrufe und Methoden fest einer Klasse zu:
- statische Methoden
    - kein Zielobjekt, richten sich an eine ganze Klasse
    - ohne Zielobjekt kein dynamischer Typ, keine Entscheidungsgrundlage für dynamisches Binden
- Konstruktoren
    - kein Zielobjekt, der Konstruktor soll ja erst eines liefern (s.o.)
- private Methoden
    - außerhalb der eigenen Klasse nicht sichtbar. Stehen überhaupt nicht zur Wahl.
    - private Methoden können zwar in abgeleiteten Klassen neu definiert werden, das ist aber keine Redefinition!

# Bindung von Objektvariablen

- Objektvariablen werden immer statisch gebunden
- Der Compiler legt beim Übersetzen endgültig fest, welche Objektvariablen benutzt werden
    - der statische Typ einer Variablen ist entscheidend!

# Bindung von Objektvariablen

```
public class Basisklasse{
    public int daten = 1;
}
public class Abgeleitet extends Basisklasse {
    public int daten = 2;
}
...
Basisklasse x = new Abgeleitet ();
System.out.println(x.daten); // gibt 1 aus
...
```

- statischer Typ von x ist Basisklasse, deren Objektvariable wird ausgegeben
- nur wichtig bei Redefinition von Objektvariablen
    - Unabhängig davon werden Objektvariablen vererbt
    - falls nicht private

# Konstruktor-Aufrufe

- jeder Konstruktor einer abgeleiteten Klasse muss zuerst einen Basisklassen-Konstruktor aufrufen
- Folge: Das Basisklassenobjekt ist vollständig initialisiert, wenn ein abgeleiteter Konstruktor abläuft
- Voreinstellung: Default-Konstruktor der Basisklasse
- Beispiel: Konstruktor von BeschraenkterZaehler ruft automatisch den Default-Konstruktor von Zaehler auf:

```
public class BeschraenkterZaehler extends Zaehler {
   ...
   BeschraenkterZaehler (int grenze){
      // Automatischer Aufruf von Zaehler()
      this.grenze = grenze;
   }
   ...
```

# Expliziter Aufruf des Basisklassen-Konstruktors

- Der Basisklassen-Default-Konstruktor kann explizit aufgerufen werden mit super();
- Beispiel: äquivalent zum vorhergehenden:

```
public class BeschraenkterZaehler extends Zaehler{
    ...
    BeschraenkterZaehler (int grenze){
        super(); // Expliziter Aufruf von Zaehler()
        this.grenze = grenze;
    }
    ...
```

- Einschränkungen des Aufrufs von super():
  - nur ein Aufruf im Konstruktor
  - Aufruf muss erste Anweisung im Konstruktor-Rumpf sein

# Beispiel: Zähler mit Rücksetzen

- Beispiel: Klasse SchleifenZaehler
    - Zähler laufen bis zum Grenzwert, springen dann aber auf 0 zurück
- Ableiten von BeschraenkterZaehler
    - Methode erhoehen() erneut redefinieren:

```
public class SchleifenZaehler extends BeschraenkterZaehler {

   public SchleifenZaehler(int grenze) {
      super(grenze);
   }

   @Override
   public void erhoehen( ) {
      if (getWert( ) == getGrenze( )) {
         reset( );
      } else {
         super.erhoehen( );
      }
   }
}
```

# Problem: Fehlender Default-Konstruktor

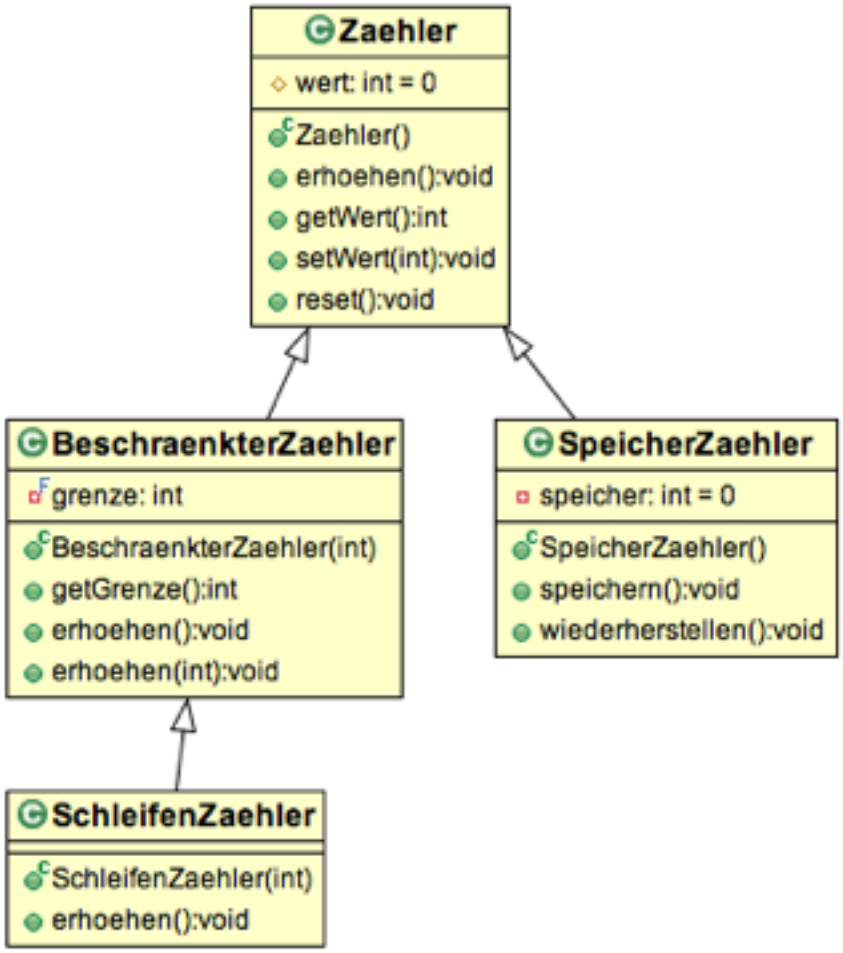
- Basisklasse SchleifenZaehler hat keinen Default-Konstruktor
  - super() kann nicht aufgerufen werden, weder implizit noch explizit
- Lösung: super() mit Argumentliste ruft den passenden Basisklassen-Konstruktor auf!

```
public SchleifenZaehler (int grenze) {
   super(grenze);
}
```

# Übersicht: Zähler-Typen

| Klasse | Zaehler | BeschraenkterZaehler | SchleifenZaehler |
|---|---|---|---|
| Konstruktor | automatisch | BeschraenkterZaehler(int) | SchleifenZaehler(int) |
| Objekt-variablen | - wert:int | kein Zugriff | kein Zugriff |
| | | - grenze:int | kein Zugriff |
| Methoden | + erhoehen(): void | + erhoehen(): void | + erhoehen(): void |
| | # setWert(int): void | ererbt | ererbt |
| | + getWert(): int | ererbt | ererbt |
| | + void reset() | ererbt | ererbt |
| | | + getWert():int | ererbt |

# UML-Übersicht



# Bezug auf die Basisklasse

- super in normalen Methoden referenziert das Basisklassenobjekt als Zielobjekt
    - weitere Nutzung von super, unabhängig vom Aufruf eines Basisklassen-Konstruktors
- super startet die Suche nach einer passenden Methode
    - dynamisches Binden in der Basisklasse, statt in der eigenen Klasse
- nur interessant für redefinierte Methoden!
- keine Verkettung von super
    - spricht nur das unmittelbare Basisklassenobjekt an
    - kann die Basisklasse der Basisklasse nicht erreichen
- Beispiel
    - redefiniertes erhoehen() von SchleifenZaehler mit explizitem Aufruf der Basisklassenmethode erhoehen().

# Expliziter Aufruf einer Basisklassenmethode

```
public class SchleifenZaehler extends BeschraenkterZaehler {
   …
   public void erhoehen() {
      if (getWert() == getGrenze()) {
         reset();
      } else {
         super.erhoehen();
      }
      ...
   }
```

- super wäre im Beispiel unnötig für getWert(),getGrenze(), reset():
  - dynamisches Binden trifft, mit und ohne super, auf die gleichen Definitionen
  - weil nicht redefiniert, sondern ererbt

# Rückgabe des eigenen Objektes

- Methode erhoehen() liefert nichts zurück (siehe Zaehler):

```
void erhoehen(){
    wert++;
}
```

- Alternative: sich selbst (= this, eigenes Objekt) zurückliefern

```
Zaehler erhoehen () {
    wert++;
    return this;
}
```

- ermöglicht Kettenaufrufe in einer Anweisung:

```
Zaehler zaehler = new Zaehler();
zaehler.erhoehen().erhoehen().erhoehen(); // 3x hochzählen
```

- Statt void das eigene Objekt zurückgeben
    - Methode flexibler einsetzbar, Beispiel: StringBuilder

# Kompatible Ergebnistypen

- Redefinition von Methoden mit kompatiblen Ergebnistypen ist möglich

# Kompatible Ergebnistypen

- Beispiel: redefinierte Fassungen von erhoehen() mit Rückgabe des eigenen Objekts

```
class Zaehler {
    Zaehler erhoehen (){
            … } }

class BeschraenkterZaehler extends Zaehler {
    BeschraenkterZaehler erhoehen(){
            … } }

class SchleifenZaehler extends BeschraenkterZaehler {
    SchleifenZaehler erhoehen(){
            … } }
```

# Übung: DoppelZaehler

- Schreiben Sie eine Klasse DoppelZaehler, die von Zaehler erbt und ihren Wert immer in Zweierschritten erhöht.
- Die Klassen soll erhoehen überschreiben und dabei die Version von erhoehen der Klasse Zaehler verwenden.
- Die Klassen soll eine Methode doppeltErhoehen bieten, die ebenfalls in Zweierschritten erhöht und eine Verkettung mehrerer Aufrufe erlaubt.

# Zusammenfassung

- Vererbung
- Methoden
- Konstruktor und Objektvariablen