HAW Hamburg
Fakultät TI
Technik und Informatik

Programmiermethodik 1
Programmiertechnik

Vererbung

Wiederholung

- Einführung
- Dynamisches Binden
- Arbeiten mit Interfaces
- Vererbung: Einführung

Ausblick für heute

Use Cases

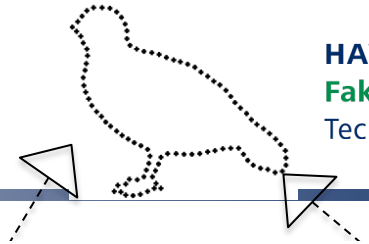
- Ich möchte eine Methode hinzufügen, die etwas ähnliches macht wie eine bestehende Methode. Die Methode soll aber andere Parameter haben.
- Ich möchte explizit auf Funktionalität aus einer Basisklasse zugreifen.
- Ich brauche eine Mischung aus einem Interface (Schnittstelle) und einer vollständigen Basisklasse (mit Implementierung).

Agenda

- Vererbung
- Methoden
- Konstruktor und Objektvariablen
- Abstrakte Basisklassen

Vererbung

Übung: Dynamische Bindung



Was ist die Ausgabe von ...?

```
public interface Bird {
    public void fly();
    public void sing();
}
public class Penguin implements Bird {
    public void fly() {
          System.out.println("Can't fly :-("); }
    public void sing() {
          System.out.println("tröt, tröt"); }
}
public class Duck implements Bird {
    public Duck() {
          System.out.println("I am duck!"); fly(); }
    public void fly() { System.out.println("flap, flap"); }
    public void sing() { System.out.println("quak, quak"); }
}
public class RubberDuck extends Duck {
    public RubberDuck(){ System.out.println("I am rubber duck!"); }
    public void fly() { super.fly();
                  System.out.println("Oh, I forgot,       can't fly"); }
    public void sing() { System.out.println("quitsch"); }
}
```

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

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

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

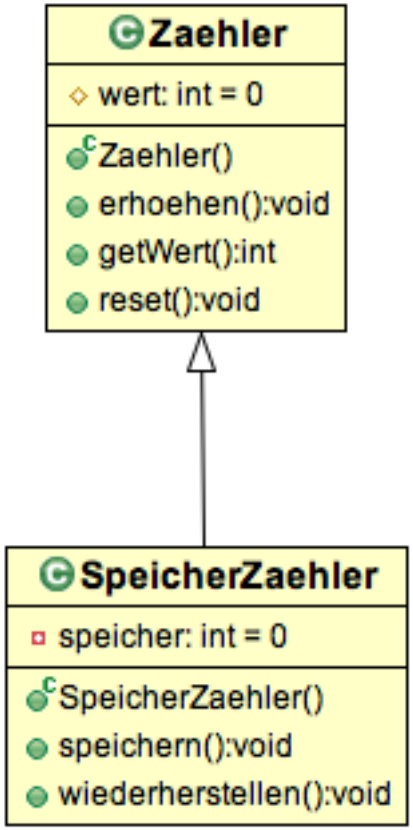
Zugriffsschutz

- 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



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


```java
/**
 * Ein beschränkter Zähler verhält sich wie ein Zähler, der aber eine Obergrenze
 * für seine Werte hat.
 */
public class BeschraenkterZaehler extends Zaehler {

  /**
   * Grenzwert.
   */
  private final int grenze;

  /**
   * Konstruktor.
   */
  public BeschraenkterZaehler(int grenze) {
    this.grenze = grenze;
  }

  /**
   * Getter.
   */
  public int getGrenze() {
    return grenze;
  }
```

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 - 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:

```
    /**
     * Erhöht den Zähler in einer gegebenen Schrittweite.
     */
    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 zaehler = new BeschraenkterZaehler(5);
for (int i = 0; i < 10; i++) {
  zaehler.erhoehen();
  System.out.format("%d ", zaehler.getWert());
}
System.out.println();
```

- Ausgabe

  1 2 3 4 5 5 5 5 5 5

Übung:

- 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 Methodenbezeichner

```
/**
 * Ein einfacher Simulator für eine KaffeeMaschine.
 */
public class KaffeeMaschine {

  /**
   * Macht Kaffee
   *
   * @param mengeKaffee
   *          Kaffeepulver in Gramm.
   */
  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:

```
/**
 * Ein SchleifenZaehler beginnt wieder von vorne, wenn er seine
Grenze erreicht
 * hat.
 */
public class SchleifenZaehler extends BeschraenkterZaehler {

  /**
   * Konstruktor.
   */
  public SchleifenZaehler(int grenze) {
    super(grenze);
  }

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

Problem: Fehlender Default-Konstruktor

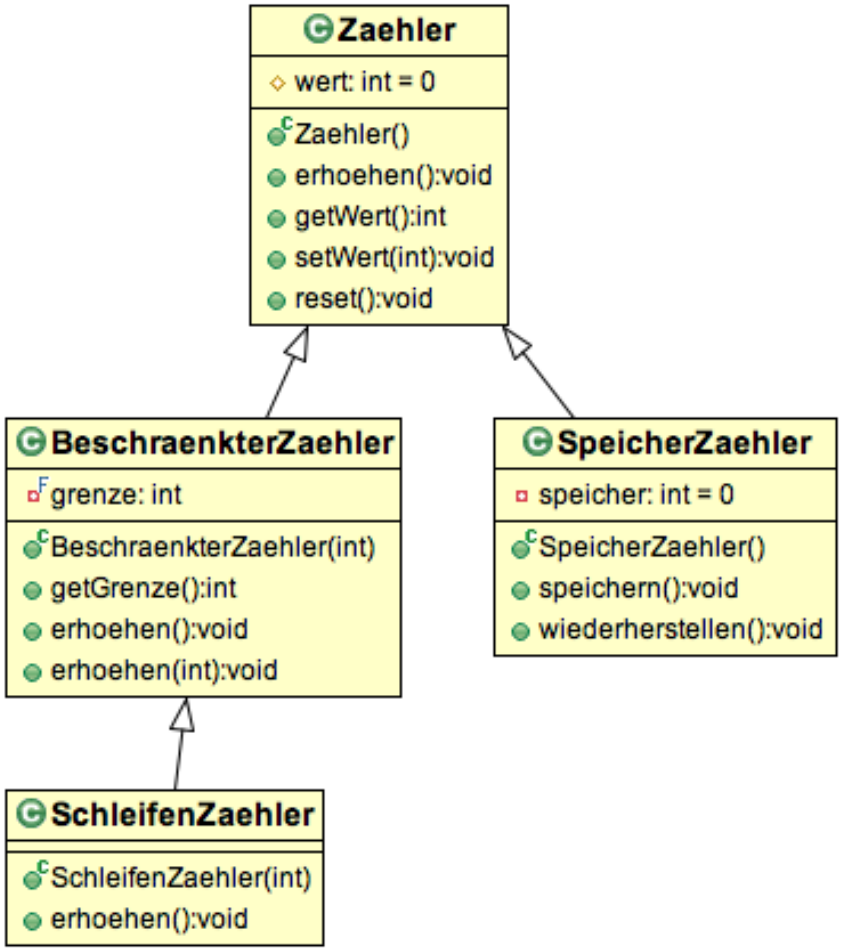
- Basisklasse `SchleifenZaehler` hat keinen Default-Konstruktor
  - `super()` kann nicht aufgerufen werden, weder implizit noch explizit
- Lösung: `super()` mit Argumentliste ruft den passenden Basisklassen-Konstruktor auf!

```
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.

Abstrakte Basisklassen

Abstrakte Basisklassen

- Bisher:
  - Interfaces
    - ausschließlich Methodenköpfe, keine Methodenrümpfe, keine Konstruktoren, keine Objektvariablen
  - Konkrete Basisklassen
    - vollständig mit Methodenrümpfen, Konstruktoren, Objektvariablen
- Mittelweg: Abstrakte Basisklassen *(engl. abstract base classes)*
- Definition mit Modifier

```
abstract class ...
```

- Methoden in einer abstrakten Basisklasse sind wahlweise …
  - konkret: mit Rumpf (wie in konkreten Klassen), oder
  - abstrakt: nur Signatur (wie bei Interfaces), statt Rumpf nur „;“

Beispiel: Abstrakter Zähler


```java
/**
 * Abstrakte Variante des Zaehlers. Nicht alle Methoden werden
implementiert.
 * Keine Instanziierung möglich.
 *
 */
public abstract class AbstrakterZaehler {

  /**
   * Aktueller Zählerstand.
   */
  protected int wert = 0;

  /**
   * Getter.
   */
  public int getWert() {
    return wert;
  }

  /**
   * Zurücksetzen des Zählers auf 0.
   */
  public void reset() {
    wert = 0;
  }

  /**
   * Erhöhen des Zählers. Diese Methode wird erst in den
abgeleiteten Klassen
   * implementiert.
   */
  public abstract void erhoehen();
}
```

Ableiten einer abstrakten Basisklasse

- eine abstrakte Basisklasse ...
  - ist unvollständig, wie ein Interface
  - dient lediglich zum Ableiten
  - kann nicht eigenständig instanziiert werden
    - nur über Objekte abgeleiteter Klassen
- abgeleitete Klassen müssen die fehlenden (abstrakten) Methoden der abstrakten Basisklasse implementieren
  - Wenn nicht oder nicht vollzählig implementiert:
    - abgeleitete Klasse ist selbst abstrakte Basisklasse, muss noch weiter abgeleitet werden

Ableiten einer abstrakten Basisklasse

- Beispiel: `KonkreterZaehler` abgeleitet von `AbstrakterZaehler`:

```
public class KonkreterZaehler extends AbstrakterZaehler {
    public void erhoehen(){
        wert++;
     }
    ...
```

Vorteile einer abstrakten Basisklasse

	Interface	Abstrakte Basisklasse
Signaturen	nur public	ohne Einschränkung
Methoden	ohne Einschränkung	ohne Einschränkung
Objektvariablen	keine	ohne Einschränkung
Klassenvariablen	nur public static final	ohne Einschränkung
Konstruktoren	keine	für abgeleitete Klassen (super), oft protected
Ableitung	von Interfaces	ohne Einschränkung

Einfache und mehrfache Vererbung

- abstrakte Basisklassen mit ausschließlich abstrakten Methoden = "rein abstrakte Basisklasse"
  - *engl. pure abstract base class*
  - konzeptionell ähnlich zu Interfaces, aber kein Ersatz für Interfaces!
- eine Klasse kann …
  - … von einer direkten Basisklasse erben
    - konkret, abstrakt oder rein abstrakt
  - … beliebig viele Interfaces implementieren
- in Java wird nur einfache Vererbung unterstützt, keine mehrfache Vererbung
  - nach `extends` darf maximal eine Basisklasse angegeben werden
  - nach `implements` aber mehrere Interfaces

Blockieren der Vererbung

- in seltenen Fällen sinnvoll: aktives Verhindern der Ableitung
- Modifier `final` der ganzen Klasse erlaubt keine abgeleiteten Klassen mehr

```
public final class FinalLastWords
```

Populäres Beispiel: Klasse **`String`**

```
public class SuperString extends String // Fehler!
```

- feinere Dosierung
  - Modifier `final` verhindert Redefinition einer einzelnen Methode

```
public class Bruch {
    public final Bruch mult(Bruch r)
                ... }
```

- `final`-Klasse beendet Folge von Ableitungen
- `final`-Methode beendet Folge von Redefinitionen

Dynamische Typbestimmung: Motivation

- in manchen Fällen muss zur Laufzeit der konkrete Typ (die Klasse) eines Objekts ermittelt werden
- Beispiel (Code in irgendeiner Anwendung):

```
public void sichereWiederherstellung(Zaehler zaehler) {
        if (zaehler instanceof SpeicherZaehler) {
                // nur bei SpeicherZaehler
                ((SpeicherZaehler) zaehler).speichern();
        }
        zaehler.reset(); // alle Zähler-Typen
}
```

- Probleme:
  - `zaehler.speichern()` nicht möglich für regulären `Zaehler`

Typprüfung mit instanceof

- zweistelliger Operator `instanceof` prüft, ob das Objekt `x` vom Typ `T` ist:

  ```
  x instanceof T
  ```

- Ergebnis:
  - `true: x` ist kompatibel zu `T` (Instanz der Klasse `T` oder einer abgeleiteten Klasse oder Implementierung des Interfaces `T`)
  - `false` ansonsten
- `instanceof` testet den
  - dynamischen Typ: Laufzeittyp
  - nicht den statischen Typ: gemäß Definition, Sicht des Compilers

Lösung durch Typecast


```
public void sichereWiederherstellung(Zaehler zaehler) {
        if (zaehler instanceof SpeicherZaehler) {
                // nur bei SpeicherZaehler
                ((SpeicherZaehler) zaehler).speichern();
        }
        zaehler.reset(); // alle Zähler-Typen
}
```

- Typecast unschön, aber harmlos: Vorangegangener Test stellt Zieltyp sicher
- Klammern nötig wegen Operatorenvorrang
  - Methodenaufruf bindet stärker als Typecast
  - `(SpeicherZaehler) zaehler.speichern();` ✗
  - `((SpeicherZaehler) zaehler).speichern()` ✔

Übung: RollenspielCharakter

- Schreiben Sie eine abstrakte Klasse `RollenspielCharakter`. Jeder `RollenspielCharakter` hat einen Namen, der im Konstruktor gesetzt wird. Jede `RollenspielCharakter` kann außerdem kämpfen, allerdings kämpfen die unterschiedlichen Charaktere sehr unterschiedlich (keine Implementierung).
- Schreiben Sie eine Klasse `Elf` (ist auch ein `RollenspielCharakter`). Beim Kämpfen schießt ein `Elf` einen Pfeil.
- Ein `Elf` kann außerdem einen Zauberspruch sagen.
- Schreiben Sie ein Code-Snippet, bei der für einen gegebenen `RollenspielCharakter`, die Kampf-Methode aufgerufen wird. Falls es sich bei dem Charakter um einen `Elf` handelt, wird außerdem ein wenig gezaubert.

Zusammenfassung

- Vererbung
- Methoden
- Konstruktor und Objektvariablen
- Abstrakte Basisklassen