Inhalt der Vorlesung

1. Klassen und Objekte
2. Vererbung
3. Enums, Wrapper und Autoboxing
4. Interfaces
5. Generics
6. Exceptions
7. Polymorphismus
8. Grafische Oberflächen
9. Streams und Lambda Expressions
10. Leichtgewichtige Prozesse – Threads

Kapitel 2: Vererbung

Inhalt

Kapitel 2: Vererbung

Lernziele

[LZ 2.1] Das Java-Vererbungskonzept erläutern können
[LZ 2.2] Das Prinzip der Aggregation erläutern können
[LZ 2.3] Die für Vererbung benötigten Sichtbarkeitsstufen anwenden können
[LZ 2.4] UML Klassenmodelle mit Vererbungsbeziehungen erstellen können
[LZ 2.5] Verstehen, wo geerbte Eigenschaften genutzt werden können
[LZ 2.6] Die Schlüsselworte „this“ und „super“ anwenden können
[LZ 2.7] Konstruktorenverkettung verstehen und anwenden können
[LZ 2.8] Die Klasse Object und ihre Methoden kennen
[LZ 2.9] Den Operator „instanceof“ kennen und anwenden können
[LZ 2.10] Das Schlüsselwort „final“ und seine Wirkung kennen
[LZ 2.11] Abstrakte Klassen und Methoden kennen und ihren Einsatz motivieren können
[LZ 2.12] Wissen, wann Aggregation und wann Vererbung genutzt werden sollte

2. Vererbung

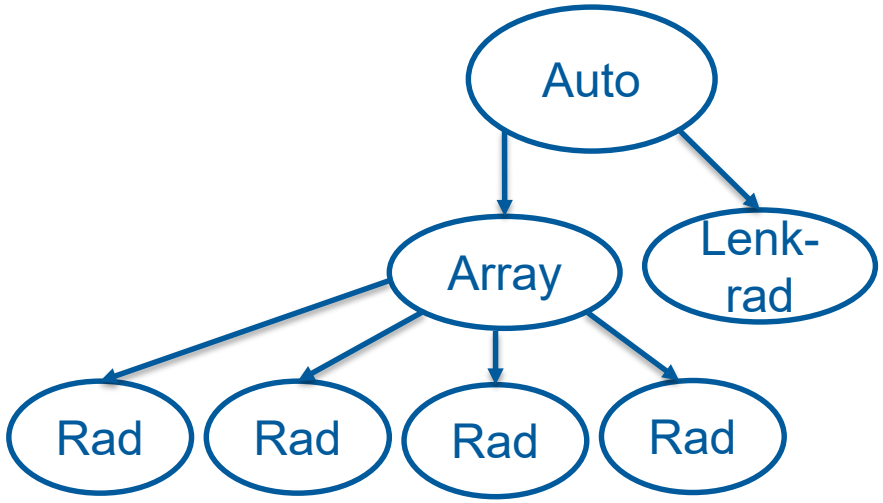
2.1 Motivation und Konzept

Bei der Entwicklung großer Software-Systeme ist es aus wirtschaftlichen Gründen wichtig, existierenden Quellcode wiederzuverwenden. In Java gibt es hierfür prinzipiell 2 Möglichkeiten: Aggregation und Vererbung

Aggregation

Ein Objekt ist häufig aus anderen Objekten zusammengesetzt. Die natürlichsprachliche Formulierung „Ein Auto hat vier Räder und ein Lenkrad“ kann in Java als Auto-Objekt mit Referenzen auf 4 Rad-Objekte und 1 Lenkrad-Objekt umgesetzt werden. Auf Klassenebene definiert man die Klasse Auto entsprechend mit einem Attribut vom Typ Rad-Array und einem Attribut vom Typ Lenkrad:

```
class Auto {
        private Rad[] raeder = new Rad[4];
        private Lenkrad lenkrad = new Lenkrad();
        …
}
```

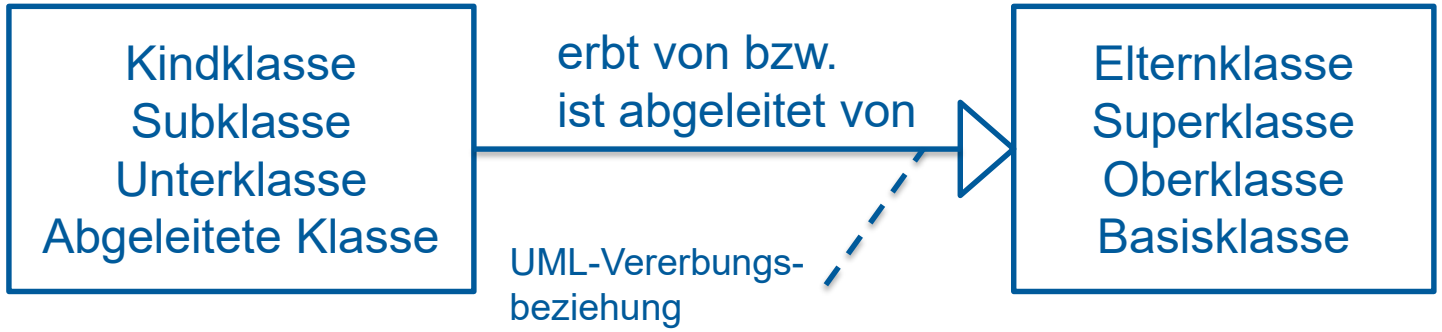


Diese Art der Wiederverwendung bestehender Klassen nennt man Aggregation.

Vererbung
Die zweite Art der Wiederverwendung in objektorientierten Sprachen ist die Vererbung. Erbt eine Klasse K2 von einer Klasse K1, so erhält K2 alle nicht-privaten Attribute und Methoden von K1. K2 definiert in der Regel noch weitere, „eigene“ Attribute und Methoden und stellt damit eine Spezialisierung von K1 dar. K2 ist eine von K1 abgeleitete Klasse, syntaktisch wird dies in Java mit dem Schlüsselwort „extends“ (erweitert) formuliert:

```
class K2 extends K1 { … }
```

Die Darstellung der Begriffe und Vererbungsbeziehung als UML-Diagramm:



Vererbung definiert eine **„ist-ein“-Beziehung**. Es gilt z.B. „eine Aktie **ist ein** Vermögenswert“, jedoch nicht „eine Aktie **hat** einen Vermögenswert“. Damit macht es Sinn, Aktie von Vermögenswert abzuleiten. Aggregation bietet sich nicht an, da es weniger Sinn macht, wenn ein Aktienobjekt eine Referenz auf ein Vermögenswert-Objekt besitzt.

Vererbung und Sichtbarkeitsstufen

Nur nicht-private Eigenschaften werden vererbt. Zur feineren Steuerung der Vererbung wird über das Schlüsselwort „`protected`“ eine Sichtbarkeitsstufe eingeführt, die Eigenschaften beschreibt, die nicht-öffentlich sind, jedoch trotzdem vererbt werden.

Beispiel:

```
class Person {
    protected String name;
    private Adresse adresse; // Aggregation eines Adress-Objekts
    public Adresse getAdresse() {
        return adresse;
    }
}

class Student extends Person { // Student erbt von Person
       private int matrikelnummer;
}
```

Erläuterungen zum Beispiel:

- Die Student-Klasse erbt das Attribut „name“ sowie die Methode „getAdresse“ von der Oberklasse Person.
- Die Methoden eines Student-Objekts können entsprechend die geerbten Eigenschaften sowie „matrikelnummer“ nutzen.
- Der Lesezugriff auf das Adress-Attribut ist über den vererbten Getter „getAdresse“ möglich!

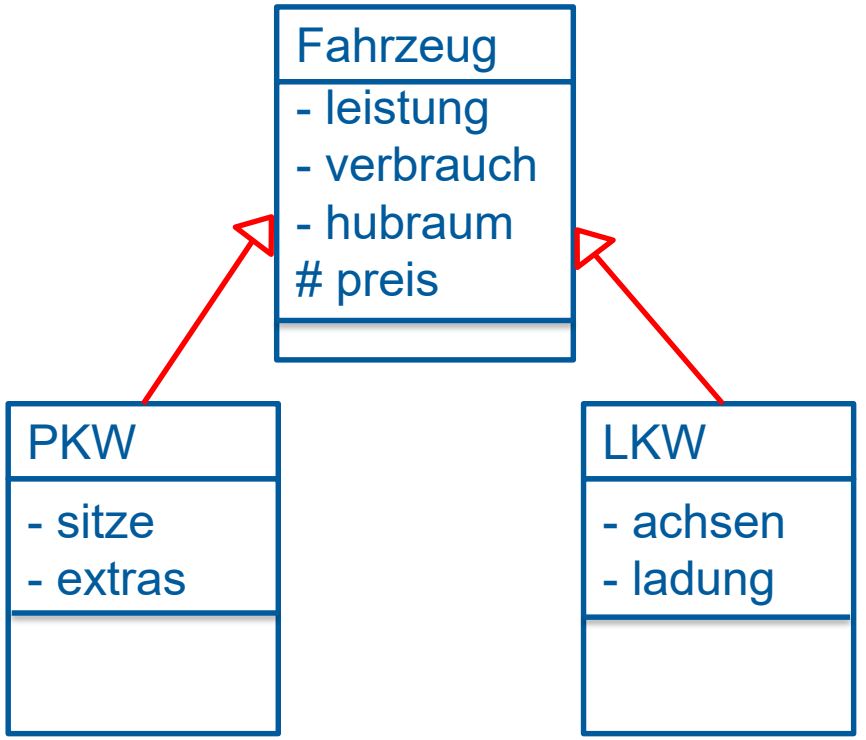
In einem UML-Klassendiagramm wird die Sichtbarkeitsstufe „protected“ durch das Zeichen „#“ (Raute) dargestellt (vgl. Beispiel auf der nächsten Seite).

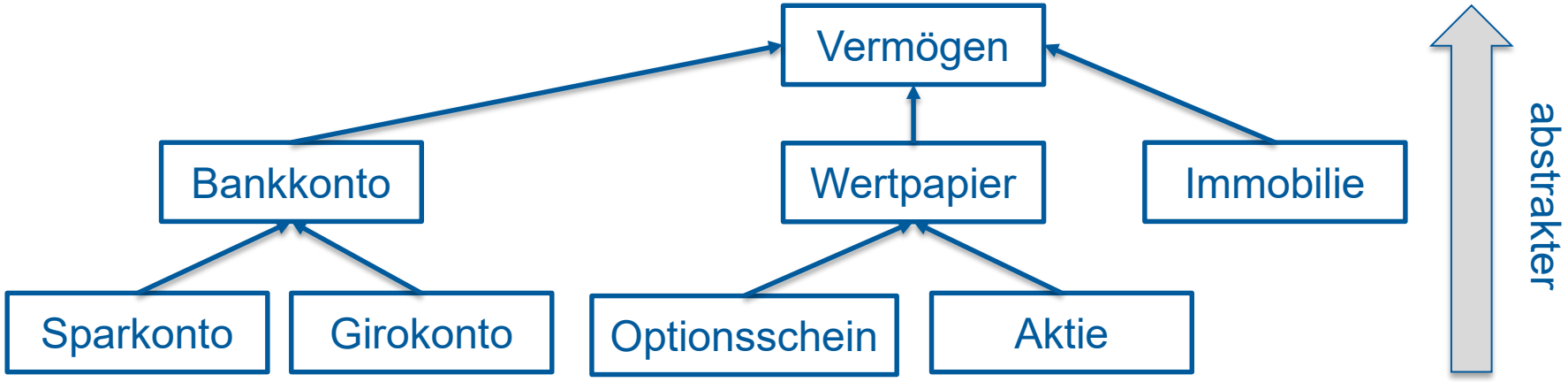
UML-Klassendiagramme und Vererbung

Die Notation für Klassenmodelle wird um die Vererbungsbeziehung erweitert – s. rote Pfeile links in der Abbildung. Die Pfeilspitze befindet sich immer an der Oberklasse!

Das Modell rechts enthält folgende Aussagen:

- PKW und LKW erben Eigenschaften der Oberklasse Fahrzeug
- Gemeinsame Attribute und Methoden (hier nicht gezeigt) werden durch die Oberklasse wiederverwendbar
- Die Klasse Fahrzeug hat 4 Attribute von denen nur preis vererbt wird, da die übrigen Attribute privat sind
- Die Klasse PKW hat 3 Attribute, ein geerbtes und 2 „lokal“ definierte:
  preis, sitze, extras
- Die Klasse LKW hat 3 Attribute, ein geerbtes und 2 „lokal“ definierte:
  preis, achsen, ladung





Vererbungshierarchien

In Java kann eine Klasse beliebige viele Unterklassen, jedoch nur eine Oberklasse haben – man spricht von Einfachvererbung. Andere objektorientierte Sprachen wie z.B. C++ erlauben dagegen Mehrfachvererbung. Durch die Einfachvererbung entsteht in der grafischen Darstellung eines Klassenmodells ein Ableitungsbaum. Die Wurzel des Baums enthält die abstrakteste Klasse, je näher an die Blätter des Baums man entlang der Beziehungspfeile gelangt, desto konkreter werden die Klassen. Ordnet man die Klassen einer Abstraktionsstufe auf der gleichen Baumhöhe an, so entsteht eine Vererbungshierarchie.

2. Vererbung

2.3 Zugriff auf geerbte Eigenschaften

Im Beispiel rechts erbt die Klasse Ableitung von der Klasse Basis folgende Eigenschaften:

- Attribute: `a, b, c`
- Methoden: `f, g, h`

d bzw. k sind `private` und werden daher nicht vererbt!

Entsprechend sind in der Funktion f der Klasse Ableitung folgende Zugriffe nicht erlaubt:

- `d = 4;`
- `k();`

Beachte:

- f ruft sich selbst rekursiv auf, was zu einer Endlosschleife führt! Mit `super.f();` kann die überschriebene Methode der Oberklasse aufgerufen werden, s. auch Abschnitt 2.4
- Der Schreibzugriff auf `d` könnte durch einen öffentlichen Getter in der Klasse Basis ermöglicht werden

```
public class Basis {
    public int a;
    protected int b;
    int c;
    private int d;

    public void f() { … };
    protected void g() { … };
    void h() { … };
    private void k() { … };
}

public class Ableitung extends Basis {
    public void f() {
        a = 1;
        b = 2;
        c = 3;
        d = 4;
        f();
        g();
        h();
        k();
}}
```

„this“ und „super“
Jedes Java-Objekt besitzt die vordefinierten Referenzen „this“ und „super“. „this“ verweist auf das Objekt selbst, während „super“ eine Referenz auf die Oberklasse liefert.
Beachte: super ist nur mit einem konkreten Zugriff nutzbar, also in der Form `super.oberklassenMethode()` bzw. `super.oberklassenAttribut`. „`this`“ dagegen kann auch für sich genutzt werden, um z.B. einen Methodenaufruf mit dem aktuellen Objekt zu parametrisieren: `eineMethode(this <, weitere Parameter>);`

Beispiel für die Nutzung von `super`:

```
class Auto { public String toString() { ... } }

class Cabrio extends Auto {
        public String toString() {
                return "Cabrio - " + super.toString();
        }
}
```

`super` wird hier genutzt, um die aufgrund der Namensgleichheit überschriebene `toString-Methode` der Oberklasse aufzurufen. Ohne `super` würde der `toString-Aufruf` zu einer endlos-Rekursion führen, da stets die lokale Methode aufgerufen werden würde!

Konstruktoren in abgeleiteten Klassen

Ein Konstruktor (CTOR) hat die Aufgabe, ein Objekt im Rahmen der Objekterzeugung korrekt zu initialisieren. Bei Erzeugung eines Objekts einer abgeleiteten Klasse müssen die Konstruktoren entlang der Vererbungshierarchie von oben nach unten ausgeführt werden, um eine korrekte Initialisierung der geerbten Attribute zu gewährleisten. Andernfalls könnte z.B. eine nicht-initialisierte Oberklassen-Referenz für einen Methodenaufruf genutzt werden, was zu einer `NullPointerException` führen wurde, da Referenzen stets mit `null` initialisiert werden.
Um obige Vorgehensweise zu gewährleisten, muss jeder Konstruktor einer abgeleiteten Klasse in der ersten Zeile einen geeigneten (d.h. mit der selben Parameterliste) Oberklassen-Konstruktor aufrufen:

```
class Oberklasse {
        public Oberklasse() { … }
}

class Ableitung extends Oberklasse {
        public Ableitung() {
                super(); // Aufruf des Oberklassen-CTORS in der 1. Zeile!
                …
        }
}
```

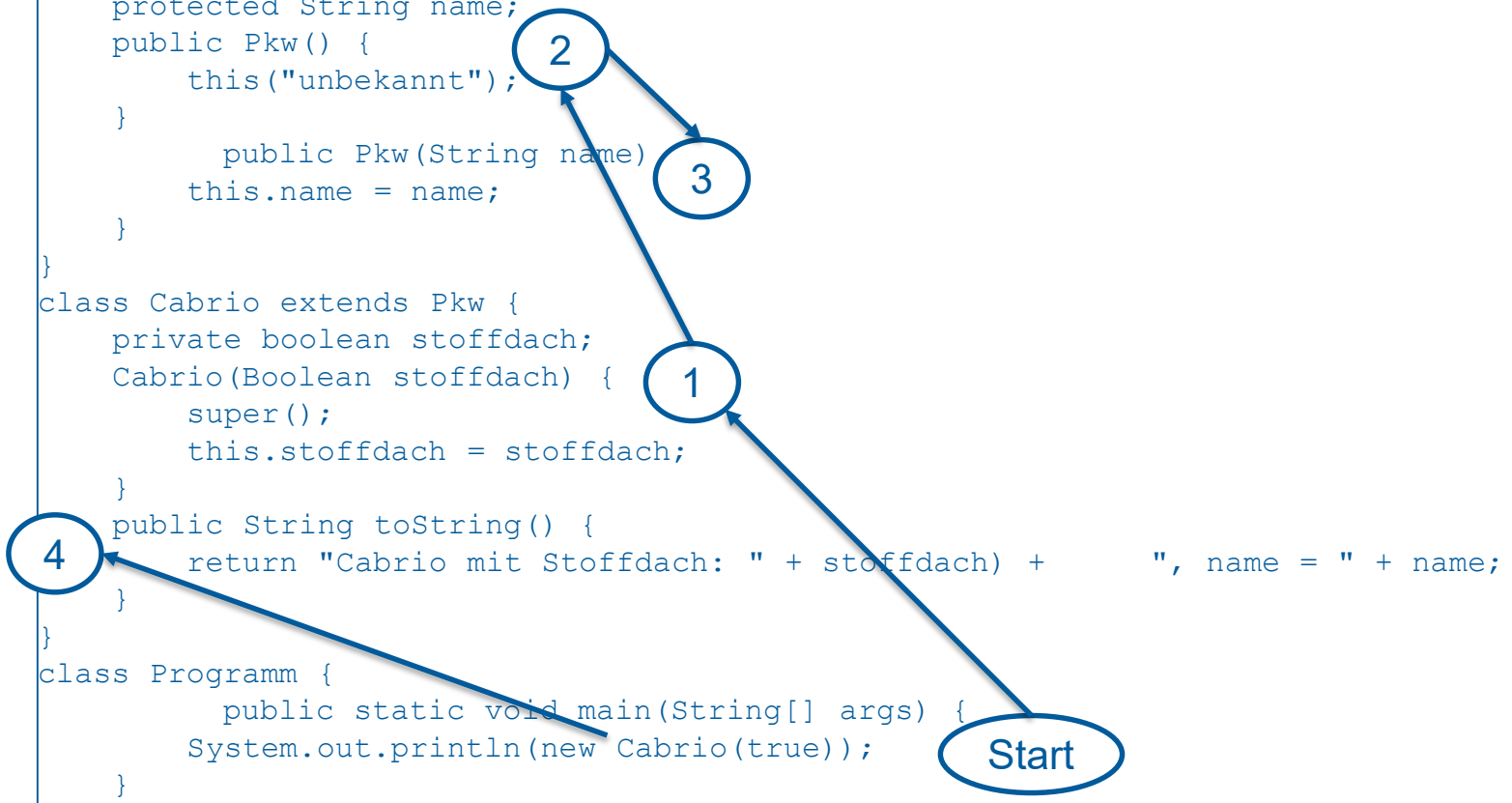
Solange kein benutzerdefinierter CTOR existiert, erzeugt der Compiler einen parameterlosen CTOR, um die Konstruktorenverkettung zu gewährleisten. Ebenso ergänzt der Compiler in jedem benutzerdefinierten Konstruktor – falls nötig – einen super-Aufruf als erste Zeile.

Das Beispiel auf der nächsten Seite zeigt die Verwendung von „this“ und „super“:

- Die Zahlen verdeutlichen, in welcher Reihenfolge das Programm ausgeführt wird.
- Zunächst wird das Cabrio-Objekt initialisiert (Schritte 1-3), dann wird die `toString`-Methode aufgerufen (Schritt 4, implizites `toString()`, da `println()` einen String erwartet).
- Der jeweils „passende“ CTOR wird anhand der Parameter beim Aufruf ermittelt.
- `this("unbekannt")` bewirkt den CTOR-Aufruf innerhalb der gleichen Klasse.

Ausgabe: `Cabrio mit Stoffdach: true, name = unbekannt`

```
class Pkw {
    protected String name;
    public Pkw() {
        this("unbekannt");
    }
      public Pkw(String name)
        this.name = name;
    }
}
class Cabrio extends Pkw {
    private boolean stoffdach;
    Cabrio(Boolean stoffdach) {
        super();
        this.stoffdach = stoffdach;
    }
    public String toString() {
        return "Cabrio mit Stoffdach: " + stoffdach) +      ", name = " + name;
    }
}
class Programm {
      public static void main(String[] args) {
        System.out.println(new Cabrio(true));
    }
}
```



Die Klasse java.lang.Object ist die Basisklasse für alle Klassen, da der Compiler implizit folgende Vererbungsbeziehung ergänzt:

```
class MeineNeueKlasse extends Object { … }
```

Erbt eine Klasse von einer anderen Klasse, so erbt sie indirekt auch von Object, da geerbte Eigenschaften beliebig tief vererbt werden. Diese Vorgehensweise sorgt dafür, dass jedes Objekt gewisse Standard-Methoden von Object erbt.
Unter anderem sind dies:

- boolean equals(): Objektvergleich
- String toString(): liefert Stringrepäsentation als <Klassenname>@<hashcode>
- Class getClass(): liefert Klassenbeschreibung des Objekts
- int hashcode(): liefert einen Hashwert für das Objekt zur Speicherung in Hashtabellen (s. Kapitel 6)

In der Regel müssen die Methoden in benutzerdefinierten Klassen überschrieben (=redefiniert) werden. equals vergleicht z.B. in der Default-Implementierung nicht die Inhalte, sondern die Referenzen der zu vergleichenden Objekte.

2. Vererbung

2.6 Überschreiben geerbter Methoden

Geerbte Methoden können durch Angabe einer Methode mit der selben Signatur (=Parametertypen + Ergebnistyp) überschrieben, d.h. redefiniert werden.
Mittels super.<Methode>(<Parameter>) kann die überschriebene Methode weiterhin aufgerufen werden. Im Beispiel unten ruft main BuntesQuadrat.toString() und BuntesQuadrat.toString() ruft Quadrat.toString() auf:

```
public class Quadrat {
    private double seitenLaenge;
    Quadrat(double seitenlaenge) {
        this.seitenLaenge = seitenlaenge;
    }
    public String toString() {
        return "seitenLaenge=" + seitenLaenge;
    }
    public static void main(String args[]) {
        System.out.println(new BuntesQuadrat(2.25, "blau").toString());
    }}
class BuntesQuadrat extends Quadrat {
    private String farbe;
    public BuntesQuadrat(double pSeitenlaenge, String farbe) {
        super(pSeitenlaenge); this.farbe = farbe;
    }
    public String toString() {
        return "BuntesQuadrat [farbe=" + farbe + ", "
            + super.toString() + "]";
}}
```

2. Vererbung

2.7 Typprüfung mit „instanceof“

In vielen Situationen erhält man eine Objektreferenz (z.B: als Parameter) und möchte überprüfen, von welchem genauen Typ die Referenz ist. Der Operator `instanceof` kann hier weiter helfen: er liefert `true`, falls eine Objektreferenz einem bestimmten Typ entspricht.
Eine Hauptanwendung sind sogenannte „Downcasts“, s. folgende Beispiele:

```
Object o = "abc";
if (o instanceof String) {
    System.out.println(((String) o).length()); // Downcast
}
Quadrat q = new BuntesQuadrat(1., "gelb");          // ***
System.out.println(q instanceof Quadrat);           // false
System.out.println(q instanceof BuntesQuadrat);     // true

private void gibQuadratFarbe(Quadrat q /* *** */) {
    if (q instanceof BuntesQuadrat) {
                BuntesQuadrat bQ = (BuntesQuadrat) q; //
Downcast
                return bQ.getFarbe();
    } else
                return "unbekannt";
}
```

*** Substitutionsprinzip: anstelle einer Oberklassenreferenz (hier: Quadrat) kann eine Unterklassenreferenz (hier: BuntesQuadrat) genutzt/übergeben werden.

Das Schlüsselwort „final“ kann in Java bei Klassen-, Methoden-, und Attributdefinitionen genutzt werden, um anzuzeigen, dass die jeweilige Eigenschaft nicht mehr geändert werden kann. Die genaue Bedeutung unterscheidet sich jedoch.

final bei Klassendefinitionen
final class MeineKlasse bedeutet, dass MeineKlasse nicht mehr ableitbar ist. Beispiele hierfür sind die Klassen String sowie StringBuffer. Die final-Deklaration einer Klasse ist in der Regel mit Sicheraspekten bzw. Performance begründet.

final bei Methodendefinitionen
Als final gekennzeichnete Methoden dürfen in abgeleiteten Klassen nicht überschrieben werden. Damit lassen sich bestimmte Funktionalitäten entlang einer Vererbungskette sicher stellen.

final bei Attributdefinitionen
Als final gekennzeichnete Attribute können nach der Deklaration und gleichzeitiger Initialisierung nicht mehr geändert werden. Dies gilt auch für Objektreferenzen, wobei zu beachten ist, dass das Objekt, auf das verwiesen wird, durch Methodenaufrufe durchaus seine Attributwerte ändern kann!

Beispiele:

```
class Math { /* Aussschnitt ! */
    final public double PI = 2.141592653589793;
}

final Quadrat q = new Quadrat(1.9);
q.setSeitenlaenge(2.5); // Quadratinhalt geändert!
q = null; // nicht möglich wg. final-Deklaration
```

Sehr oft möchte man Funktionalität bzw. Daten für eine Menge von Klassen per Vererbung vorgeben. Die Basisklasse, die diese Vorgaben macht, kann jedoch häufig nicht sinnvoll instanziiert werden, weil wesentliche Aspekte erst in konkreten Subklassen hinzukommen. In solchen Fällen bietet es sich an, die Basisklasse als abstrakte Klasse zu definieren:

```
abstract class ImageFile {// gibt allg. Eigenschaften vor }
```

Die Klasse Basis kann nicht instanziiert werden, d.h. `new ImageFile()` ist nicht zulässig, obwohl ggf. Konstruktoren definiert sind. Erst zu den nicht-abstrakten Subklassen können dann Objekte erzeugt werden. Beispiel:

```
class JPGFile extends ImageFile { // Spezifisches für JPEG Image  }
class PNGFile extends ImageFile { // Spezifisches für PNG Image }
```

Abstrakte Methoden

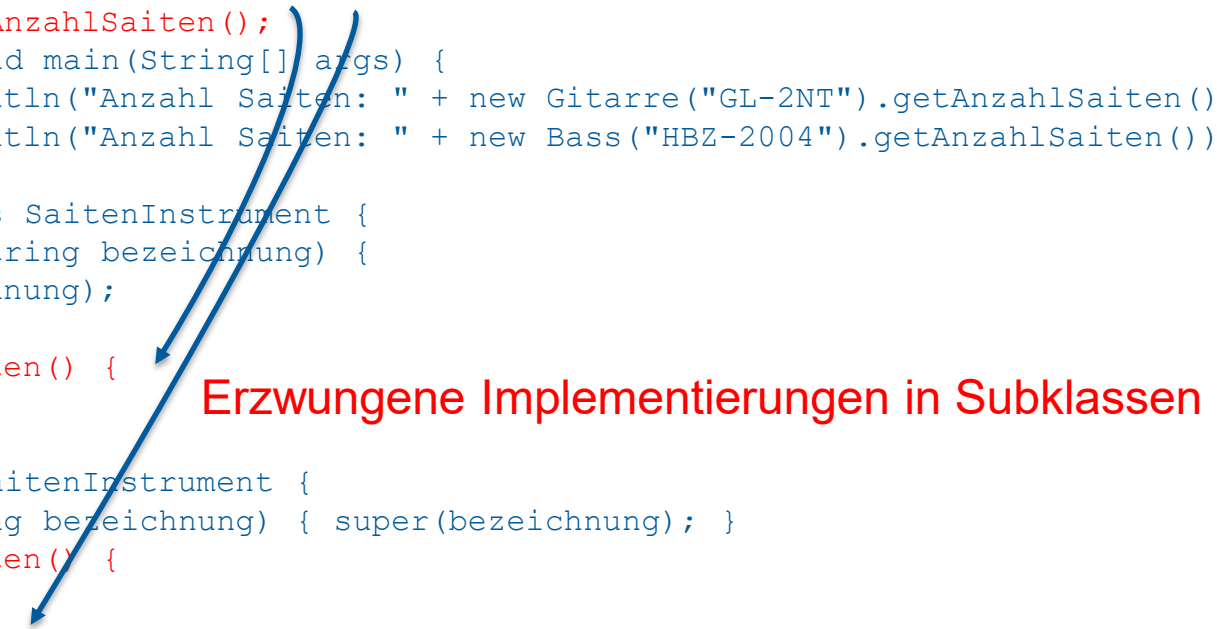
Um eine bestimmte Schnittstelle in den abgeleiteten Klassen zu erzwingen, kann eine abstrakte Klasse abstrakte Methoden vorgeben: diese gibt eine Methodensignatur vor, besitzt aber keine Implementierung. Sobald `eine` Klasse mindestens eine abstrakte Methode vorgibt, muss sie als `abstract` definiert werden, da sonst Objekte mit Methoden ohne Implementierung aufgerufen werden könnten!

Beispiel:
```
abstract class ImageFile { abstract public void display(); }
class JPGFile extends ImageFile {
    public void display() { ... // Implementierung }
}
```

Die Klasse `ImageFile` gibt die `display`-Methode vor, die konkrete Klasse `JPGFile` muss diese implementieren! Andernfalls erzeugt der Compiler eine Fehlermeldung.

Beispiel: Saiteninstrument gibt das Attribut Bezeichnung vor und fordert die Schnittstellenoperation *getAnzahlSaiten* von allen Subklassen. Entsprechend müssen Gitarre und Bass die abstrakte Methode überschreiben und sie dabei implementieren.
Vorteil: es wird garantiert, dass jedes Saiteninstrument die Anzahl Saiten liefern kann.

```
abstract public class SaitenInstrument {
    protected String bezeichnung;
    public SaitenInstrument(String bezeichnung) {
        this.bezeichnung = bezeichnung;
    }
    abstract int getAnzahlSaiten();
    static public void main(String[] args) {
      System.out.println("Anzahl Saiten: " + new Gitarre("GL-2NT").getAnzahlSaiten());
      System.out.println("Anzahl Saiten: " + new Bass("HBZ-2004").getAnzahlSaiten());
    }}
class Gitarre extends SaitenInstrument {
    public Gitarre(String bezeichnung) {
        super(bezeichnung);
    }
    int getAnzahlSaiten() {
        return 6;
    }}
class Bass extends SaitenInstrument {
    public Bass(String bezeichnung) { super(bezeichnung); }
    int getAnzahlSaiten() {
        return 4;
    }}
```



2. Vererbung

2.10 Design-Entscheidung: Vererbung oder Aggregation?

Bei der Überlegung, welche Klassen für eine Aufgabenstellung definiert werden sollen, steht man typischerweise vor der Entscheidung, Vererbung oder Aggregation einzusetzen. Grundsätzlich ist hier die Aggregation, also das Zusammensetzen einer Klasse aus anderen, vorzuziehen. Aggregation bietet mehr Möglichkeiten als Einfachvererbung und schafft weniger enge Abhängigkeiten zwischen den Klassen.

Einige einfache Regeln helfen hier bei der Design-Entscheidung:
„Klasse A hat/besteht aus einem/mehreren Klasse B“ → Aggregation

```
class A {
        private B b;
        // oder:
        private B[] bListe;
}
```

„Klasse A ist ein Klasse B“ → Vererbung

```
class A extends B { … }
```