9. Vererbung und Polymorphie

9.1 Eine Bücherliste

9.1.1 Zielsetzung

Wir wollen ein **ArrayList**-Objekt erstellen, in dem wir Objekte der Klasse **Buch** speichern können. An diesem Beispiel wollen wir dann das grundlegende Prinzip der ***Vererbung*** kennenlernen.

9.1.2 Einstieg: Die Klassen Buch und Buchliste

Die Klasse Buch

Minimal-Quelltext

```
public class Buch
{
    private String titel, autor;
    private int jahr;

    public Buch(String titel, String autor, int jahr)
    {
        this.titel = titel;
        this.autor = autor;
        this.jahr  = jahr;
    }

    public void zeige()
    {
        System.out.println("---------------------------------");
        System.out.println("Titel : " + titel);
        System.out.println("Autor : " + autor);
        System.out.println("Jahr  : " + jahr);
        System.out.println("_________________________________/");
        System.out.println();
    }
}
```

Diese erste Version der Klasse **Buch** ist noch recht einfach aufgebaut: Drei Instanzvariablen, ein Konstruktor, der diese Instanzvariablen initialisiert, und eine Methode `zeige()`. Es handelt sich hier um einen Minimal-Quelltext. Im nächsten Schritt wollen wir Getter-Methoden und überprüfende Setter-Methoden ergänzen.

Checkende Setter-Methoden

Beispiel setAutor()

```
public void setAutor(String autor)
{
   if (autor == null || autor.isBlank())
      throw new IllegalArgumentException("Der Autor darf nicht leer sein.");
   this.autor = autor;
}
```

Die Setter-Methode für den Titel sieht ähnlich aus, und bei der Setter-Methode für das Jahr müssen wir die Jahreszahl überprüfen, ob sie gültig ist.

Mit solchen kontrollierenden Setter-Methoden können wir auch den Konstruktor der Klasse etwas umgestalten:

```
public Buch(String titel, String autor, int jahr)
{
   setTitel(titel);
   setAutor(autor);
   setJahr(jahr);
}
```

Getter-Methoden

Beispiel getAutor()

```
public String getAutor()
{
   return autor;
}
```

Bei den Getter-Methoden müssen wir keine weiteren Prüfungen durchführen, daher sind sie recht einfach gestaltet.

Vollständiger Quelltext

Den vollständigen Quelltext mit Getter- und überprüfenden Setter-Methoden finden Sie auf der folgenden Seite: https://u-helmich.de/inf/OOP-Einfuehrung/QT/09/91/Buchliste-1-erweitert/Buch.java

Die Klasse Buchliste

Quelltext (einfach)

```
import java.util.ArrayList;

public class Buchliste
{
    private ArrayList<Buch> liste;

    public Buchliste()
    {
        liste = new ArrayList<>();

        erzeuge();
        zeige();
    }

    public void erzeuge()
    {
        liste.add(new Buch("Einführung in Java","Meier, Otto",2025));
        liste.add(new Buch("Lexikon der Informatik","Duden-Verlag",2012));
        liste.add(new Buch("Der Herr der Ringe","J.R. Tolkien",1964));
        liste.add(new Buch("Prinzipien der OOP","Müller, Jonathan",2020));
    }

    public void zeige()
    {
       for (Buch b: liste)
       {
          b.zeige();
       }
    }


    public static void main(String[] args)
    {
       new Buchliste();
    }
}
```

Die Verwendung von `ArrayList<Buch>` stellt sicher, dass nur Objekte der Klasse **Buch** in dieser Liste gespeichert werden können (***Typsicherheit).***

Der Konstruktor initialisiert die Liste und ruft anschließend die Methoden `erzeuge()` und `zeige()` auf.

In der Methode `erzeuge()` werden mehrere Buch-Objekte erzeugt und der Liste hinzugefügt. Eine Schleife ist hierfür nicht sinnvoll, da die Objekte unterschiedliche Attributwerte besitzen und daher individuell konstruiert werden müssen.

Die Methode `zeige()` verwendet dagegen eine for-each-Schleife, um alle Elemente der Liste zu durchlaufen. Für jedes Buch-Objekt wird die Ausgabe an die Methode `zeige()` der Klasse **Buch** ***delegiert***. Eine Schleife ist hierfür perfekt geeignet, da alle Objekte nach dem gleichen Schema verarbeitet werden.

Wenn das Objekt `liste` vom Typ **MyArrayList** (Kapitel 8) ist, können wir bei der `zeige()`-Methode allerdings keine for-each-Schleife anwenden, da es sich bei **MyArrayList** nicht um eine offizielle Sammlungsklasse handelt, die entsprechende Möglichkeiten implementiert hat.

Die `main()`-Methode erzeugt lediglich ein Objekt der Klasse **Buchliste** und startet damit das Programm.

Quelltext (Version mit Exception-Handling)

```
import java.util.ArrayList;

public class Buchliste
{
    private ArrayList<Buch> liste;

    public Buchliste()
    {
      // wie bisher
    }

    public void erzeuge()
    {
        try
        {
            liste.add(new Buch("Einführung in Java", "Meier, Otto", 2025));
            liste.add(new Buch("Lexikon der Informatik", "Duden-Verlag", 2012));
            liste.add(new Buch("Der Herr der Ringe", "J.R. Tolkien", 1964));
            liste.add(new Buch("Prinzipien der OOP", "Müller, Jonathan", 2020));
        }
        catch (IllegalArgumentException e)
        {
            System.out.println("Fehler beim Erzeugen eines Buches:");
            System.out.println(e.getMessage());
        }
    }

    public void zeige()
    {
       // wie bisher
    }

    public static void main(String[] args)
    {
        new Buchliste();
    }
}
```

So könnte eine Version der Klasse **Buchliste** mit Exception-Handling aussehen. Ein Problem besteht allerdings noch bei der Methode `erzeuge()`: Wenn bei der Erzeugung des ersten Buchs eine Exception auftritt, wird die ganze `erzeuge()`-Methode sofort abgebrochen und der catch-Block wird ausgeführt. Die noch folgenden Bücher werden der Liste nicht mehr hinzugefügt.

Eine noch bessere Version von **Buchliste** besitzt eine eigene `fuegeBuchHinzu()`-Methode:

```
private void fuegeBuchHinzu(String titel, String autor, int jahr)
{
   try
   {
      liste.add(new Buch(titel, autor, jahr));
   }
   catch (IllegalArgumentException e)
   {
      System.out.println("Fehler beim Hinzufügen eines Buches:");
      System.out.println(e.getMessage());
   }

}
```

Die Methode `erzeuge()` ruft jetzt für *jedes* Buch diese neue Methode auf. Sollte dabei eine Exception geworfen werden, wird diese in `fuegeBuchHinzu()` nur für dieses eine Buch behandelt, die anderen Bücher können dann trotzdem hinzugefügt werden.

Den vollständigen Quelltext der erweiterten Buchliste können Sie hier herunterladen:

https://u-helmich.de/inf/OOP-Einfuehrung/QT/09/91/Buchliste-1-erweitert/Buchliste.java

Die Konsolenausgabe

```
---------------------------------
Titel : Einführung in Java
Autor : Meier, Otto
Jahr  : 2025
_________________________________/

---------------------------------
Titel : Lexikon der Informatik
Autor : Duden-Verlag
Jahr  : 2012
_________________________________/

---------------------------------
Titel : Der Herr der Ringe
Autor : J.R. Tolkien
Jahr  : 1964
_________________________________/

---------------------------------
Titel : Prinzipien der OOP
Autor : Müller, Jonathan
Jahr  : 2020
_________________________________/
```

9.1.3 Spezialisierte Buch-Klassen

Das Objekt `ArrayList<Buch> liste` bzw. `MyArrayList<Buch> liste` soll jetzt weitere Buch-Klassen speichern können, beispielsweise Objekte der Klassen **Roman** und **Sachbuch**. In diesem Kapitel 9 wird das ***Prinzip der Vererbung*** vorgestellt.

Wir wollen aber zunächst einmal sehen, wie wir eine solche heterogene Liste implementieren können, wenn wir den Mechanismus der Vererbung noch nicht kennen und benutzen. In Zukunft wollen wir eine solche Implementierung jedoch nicht mehr anwenden, daher heißt der nächste Abschnitt:

Wie man es nicht machen sollte

Eine naheliegende Lösung wäre es, die Klasse **Buch** per ***Copy & Paste*** zu duplizieren und dann in eigene Klassen wie **Roman** und **Sachbuch** umzubenennen.

Diese Klassen müssten anschließend unterschiedlich erweitert werden, zum Beispiel um eine Instanzvariable `genre` (für Romane) bzw. `fachgebiet` (für Sachbücher). Auch die Methode `zeige()` müsste in jeder Klasse speziell angepasst werden.

Ein solcher Ansatz ist zwar kurzfristig umsetzbar (und passiert in der Praxis auch recht häufig), führt aber zu erheblichen Nachteilen, wie wir später noch sehen werden.

Schauen wir uns an, was passiert, wenn man die Klasse **Buch** kopiert und dann so verändert, dass man eine neue Klasse **Roman** erhält:

```
public class Roman
{
    private String titel, autor;
    private int jahr;
    private String genre;

    public Roman(String titel, String autor, String genre, int jahr)
    {
        this.titel  = titel;
        this.autor = autor;
        this.genre  = genre;
        this.jahr   = jahr;
    }

     public void zeige()
    {
        System.out.println("---------------------------------");
        System.out.println("Titel : " + titel);
        System.out.println("Autor : " + autor);
        System.out.println("Genre : " + genre);
        System.out.println("Jahr  : " + jahr);
        System.out.println("---------------------------------");
        System.out.println();
    }
}
```

Hier wurde übrigens nur die Basis-Version der Klasse **Buch** als Kopiervorlage verwendet, also die Version ohne Getter- und checkenden Setter-Methoden. Wir wollen den Rahmen dieses Skripts ja nicht sprengen.

Die rot hervorgehobenen Zeilen sind die Änderungen, die durchgeführt werden mussten. Die Klasse **Sachbuch** erzeugen wir auf die gleiche Weise, die Instanzvariable `genre` wird dann durch eine Instanzvariable `fachgebiet` ersetzt, entsprechend müssen wir den `Konstruktor` und die `zeige()`-Methode etwas ändern.

Warum man es so nicht machen sollte

Die drei Klassen **Buch**, **Roman** und **Sachbuch** unterscheiden sich nur in wenigen Details, enthalten aber ansonsten nahezu identischen Code. Diese ***Code-Duplizierung*** sollte grundsätzlich vermieden werden.

Angenommen, Sie wollen bei der Ausgabe der Bücher die Reihenfolge der Attribute ändern. Bisher wird zuerst der Titel ausgegeben, dann der Autor und am Ende das Erscheinungsjahr. Sie wollen nun aber die Ausgabe so ändern, dass zuerst der Autor ausgegeben wird, dann der Titel und das Erscheinungsjahr.

Dann müssen Sie in jeder der drei Klassen die `zeige()`-Methode anpassen - das ist ein hoher Wartungsaufwand wegen der vorliegenden Redundanz.

Außerdem kann es passieren, dass Sie bei den Anpassungen eine der drei (oder mehr) Klassen vergessen - hohe Fehleranfälligkeit.

Ein weiteres Beispiel: Sie wollen nicht nur Autor, Titel und Erscheinungsjahr in den Objekten der drei Klassen speichern, sondern ein weiteres Attribut, beispielsweise die Seitenzahl. Dann müssen Sie dieses Attribut in jeder der drei Klassen als weitere Instanzvariable deklarieren, eine entsprechende Setter- und Getter-Methode schreiben und auch die `zeige()`-Methoden in jeder Klasse anpassen.

Wie leicht passiert es, dass man diese Änderungen bei zwei Klassen durchführt, aber bei der dritten vergisst oder auf eine andere Weise durchführt, die nicht zu den bisherigen Änderungen passt?

Das bisherige Vorgehen - das man leider in der Praxis viel zu oft findet - hat mehrere Nachteile:

- ***Redundanz***: Große Teile des Quelltexts sind identisch und mehrfach vorhanden.
- ***Wartungsaufwand***: Änderungen müssen an mehreren Stellen durchgeführt werden.
- ***Fehleranfälligkeit***: Inkonsistenzen zwischen den Klassen sind leicht möglich.

Ein weiteres Problem

Abgesehen von diesen Nachteilen ist das oben beschriebene Vorgehen mit einem weiteren Problem verbunden. In einer typisierten Liste wie

```
ArrayList<Buch> liste;
```

können nur Objekte der Klasse **Buch** gespeichert werden. Objekte der Klassen **Roman** und **Sachbuch** wären nicht kompatibel zum ***Typ-Parameter*** `<Buch>` und könnten daher nicht in der Liste gespeichert werden.

Vordergründige Lösung dieses Problems

Man könnte natürlich einfach auf ***Generics*** verzichten und schreiben:

```
ArrayList liste;
```

statt

```
ArrayList<Buch> liste;
```

So können tatsächlich Objekte unterschiedlicher Klassen wie **Buch**, **Roman** und **Sachbuch** gespeichert werden (***heterogene Liste***), jedoch gehen wichtige Vorteile verloren:

- Es gibt keine ***Typsicherheit*** mehr.
- Methoden wie `get()` liefern nur noch Objekte der Klasse **Object**.
- Ein Zugriff erfordert daher explizites ***Typecasting*** und gegebenenfalls ***Typüberprüfungen*** mit dem `instanceof`-Operator.

Wie sähe eine heterogene Buchliste ohne Generics aus?

Würde man bei der Klasse **Buchliste** auf Generics verzichten, sähe die for-Schleife der `zeige()`-Methode so aus:

```
for (int i = 0; i < liste.size(); i++)
{
    Object objekt = liste.get(i);

    if (objekt instanceof Roman)
    {
       Roman roman = (Roman) objekt;
       roman.zeige();
    }
    else if (objekt instanceof Sachbuch)
    {
       Sachbuch sachbuch = (Sachbuch) objekt;
       sachbuch.zeige();
    }
    else if (objekt instanceof Buch)
    {
       Buch buch = (Buch) objekt;
       buch.zeige();
    }
}
```

Mit

```
objekt instanceof Klasse
```

überprüft man, ob das gegebene Objekt der jeweiligen Klasse angehört. Wenn das der Fall ist, wenn also beispielsweise das aktuelle Objekt der Klasse **Roman** angehört, muss das Objekt, das von `get()` geliefert wurde, per Typecasting in ein **Roman**-Objekt umgewandelt werden:

```
if (objekt instanceof Roman)
{
   Roman roman = (Roman) objekt;
   roman.zeige();
}
```

Verzichtet man auf Generics, so gehen Typsicherheit und Komfort verloren: Beim Auslesen aus einer Liste sind dann oft `instanceof`-Prüfungen und explizites Typecasting notwendig.

"Explizites Typecasting" heißt, dass das Typecasting nicht automatisch vom Compiler durchgeführt wird, sondern von dem Entwickler bewusst eingesetzt wird. Explizites Typecasting erkennt man am Datentyp, der in runden Klammern vor dem umzuwandelnden Datentyp steht.

Etwas Theorie: Generalisierung und Spezialisierung

In der objektorientierten Programmierung kennt man zwei wichtige und eng zusammenhängende Prinzipien: ***Generalisierung*** und ***Spezialisierung***.

Generalisierung

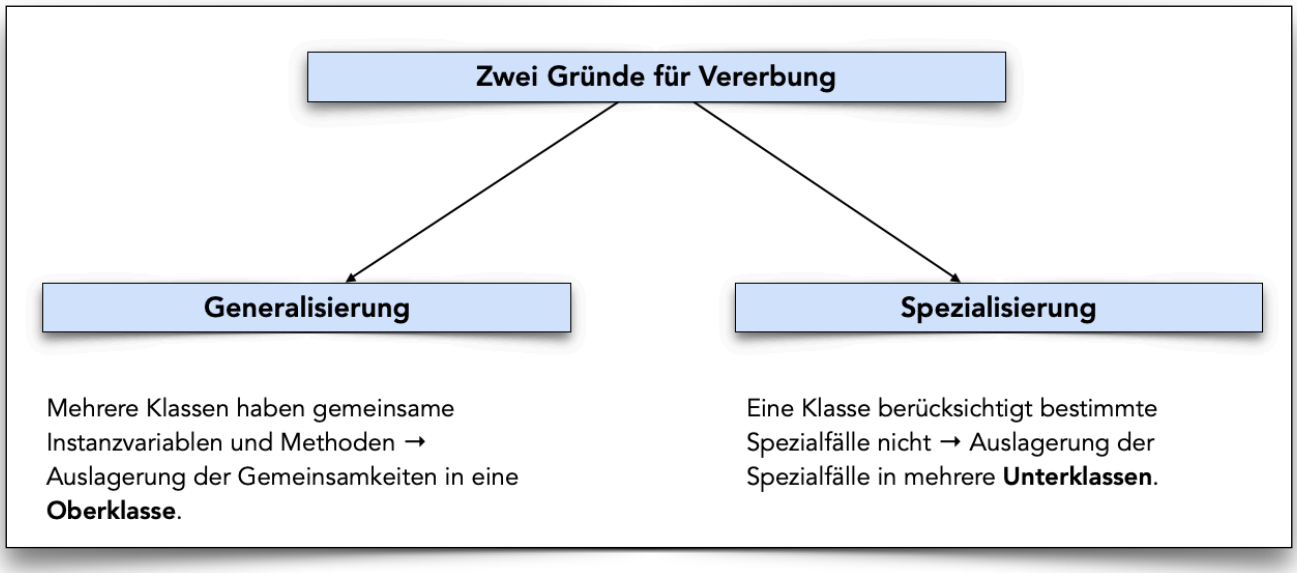
Generalisierung bedeutet, gemeinsame Eigenschaften und Verhaltensweisen mehrerer Klassen zu identifizieren und in einer ***Oberklasse*** zusammenzufassen.

In unserem Buchliste-Beispiel werden die Gemeinsamkeiten der Klassen **Roman** und **Sachbuch** – etwa `titel`, `autor` und jahr sowie die Methode `zeige()` – in die Oberklasse **Buch** ausgelagert.

Spezialisierung

Spezialisierung beschreibt den umgekehrten Weg: Ausgehend von einer allgemeinen Klasse werden spezifischere ***Unterklassen*** gebildet, die *zusätzliche* Eigenschaften (also Instanzvariablen) oder angepasstes Verhalten (also Methoden) besitzen.

Durch Spezialisierung entstehen aus der Klasse **Buch** die Unterklassen **Roman** und **Sachbuch**, die jeweils um spezifische Instanzvariablen wie `genre` bzw. `fachgebiet` erweitert werden.



9.2 Das Konzept der Vererbung

9.2.1 Unterklassen sind Spezialisten

Wir gehen jetzt - ganz im Sinne der ***Spezialisierung*** - von der Klasse **Buch** aus und erstellen zwei ***Unterklassen*** **Roman** und **Sachbuch**. Alle Gemeinsamkeiten der beiden Unterklassen haben wir in der Klasse **Buch** belassen, die dadurch zu einer gemeinsamen ***Oberklasse*** "befördert" wird.

Wichtig ist dabei: Die Methode `zeige()` der Oberklasse gibt bereits die allgemeinen Buchdaten aus. Die Unterklassen nutzen diese Implementierung über `super.zeige()` und ergänzen anschließend ihre spezifischen Informationen.

```
public class Roman extends Buch
{
    String genre;

    public Roman(String titel, String autor, int jahr, String genre)
    {
       super(titel, autor, jahr);
       this.genre = genre;
    }

    @Override
    public void zeige()
    {
       super.zeige();
       System.out.println("Genre  : " + genre);
    }
}

public class Sachbuch extends Buch
{
    String fachgebiet;

    public Sachbuch(String titel, String autor, int jahr, String gebiet)
    {
       super(titel, autor, jahr);
       this.fachgebiet = gebiet;
    }

    @Override
    public void zeige()
    {
       super.zeige();
       System.out.println("Fachgebiet  : " + fachgebiet);
    }
}
```

Grün markiert sind die zusätzlichen Instanzvariablen der Unterklassen, ihre Initialisierung und ihre Ausgabe in die Konsole.

Rot markiert sind dagegen alle Anweisungen und Anmerkungen, die sich direkt auf die Oberklasse beziehen.

Wir besprechen all diese Änderungen am Beispiel der Unterklasse **Roman**.

9.2.2 Vererbung am Beispiel der Unterklasse Roman

```
public class Roman extends Buch
{
    String genre;

    public Roman(String titel, String autor, int jahr, String genre)
    {
       super(titel, autor, jahr);
       this.genre = genre;
    }

    @Override
    public void zeige()
    {
       super.zeige();
       System.out.println("Genre  : " + genre);
    }
}
```

Die Klasse **Roman** ist eine ***Unterklasse*** von **Buch**. Das bedeutet:

- Ein Roman **IST** ein Buch (is-a-Beziehung, IST-Beziehung).
  Objekte der Klasse **Roman** besitzen also alle Instanzvariablen und Methoden der Klasse **Buch**, die nicht als `private` deklariert sind.
- Ein Roman hat **zusätzliche Eigenschaften**, die Objekte der Klasse **Buch** nicht haben. In unserem Beispiel ist das die Instanzvariable `genre`.

Das Schlüsselwort extends

Dieses Schlüsselwort bedeutet so viel wie "erbt von" oder - wörtlich übersetzt - "erweitert".

In der Tat erweitert die Klasse **Roman** die Klasse **Buch** um eine weitere Instanzvariable und eine erweiterte `zeige()`-Methode.

Das Schlüsselwort super

Im Konstruktor

Mit `super(titel, autor, jahr)` im Konstruktor der Klasse **Roman** wird der Konstruktor der Oberklasse **Buch** aufgerufen, dabei werden die Parameter `titel`, `autor` und `jahr` an diesen Oberklassen-Konstruktor weitergereicht.

Das ist notwendig, weil die Unterklasse nicht auf die privaten Attribute der Oberklasse zugreifen kann. Der ***Konstruktor*** von **Buch** kann jedoch von der Unterklasse **Roman** aufgerufen werden, er übernimmt dann die Initialisierung der privaten Instanzvariablen.

Im Konstruktor der Unterklasse **Roman** wird dann zusätzlich noch die Instanzvariable `genre` initialisiert.

Die Reihenfolge: Erst `super()`, dann weitere Initialisierungen, muss unbedingt eingehalten werden.

Der Aufruf von `super()` muss immer die erste Anweisung in einem Konstruktor der Unterklasse sein.

In der Methode zeige()

In der Methode `zeige()` von **Roman** wird mit `super.zeige()` die entsprechende Methode `zeige()` der Oberklasse **Buch** aufgerufen. Die Unterklasse **Roman** ergänzt anschließend die Ausgabe um ihre eigenen Informationen (`genre`).

Die Annotation @Override

Eine ***Annotation*** ist ein Hinweis an den Compiler, dass die folgende Methode eine Methode der Oberklasse überschreiben soll. Diese Annotation (Anmerkung) hat eine wichtige Aufgabe:

Wenn man sich in der Klasse **Roman** verschreibt, beispielsweise `ziege()` statt `zeige()`, dann würde der Compiler die falsch geschriebene Methode akzeptieren. Warum sollte die Klasse **Roman** nicht eine Methode `ziege()` haben, der Compiler weiß so etwas ja nicht.

Durch die Annotation `@Override` wird dem Compiler aber mitgeteilt, dass diese Methode `ziege()` eine *gleichnamige* Methode der Oberklasse überschreiben soll. Da es in der Oberklasse **Buch** aber keine Methode `ziege()` gibt, liegt hier offensichtlich ein Fehler vor, und der Compiler verweigert die Übersetzung.

> Unterklassen erweitern eine Oberklasse um zusätzliche Eigenschaften und Verhalten. Gemeinsame Funktionalität wird in der Oberklasse implementiert und in den Unterklassen wiederverwendet.

9.2.3 Polymorphie

Klasse Buchliste

Wir wollen nun die Klasse **Buchliste** so erweitern, dass nicht nur Objekte der Klasse **Buch** gespeichert werden können, sondern auch Objekte der beiden Unterklassen **Roman** und **Sachbuch**.

Da wir nun das Konzept der ***Vererbung*** einsetzen, wird die Erstellung einer solchen heterogenen Liste wesentlich vereinfacht, wie wir gleich sehen werden.

```
import java.util.ArrayList;

public class Buchliste
{
    private ArrayList <Buch> liste;

    public Buchliste()
    {
        liste = new ArrayList<>();

        erzeuge();
        zeige();
    }

    public void erzeuge()
    {
       liste.add(new Sachbuch("Einführung in Java","Meier, Otto",2025,"Informatik"));
       liste.add(new Sachbuch("Lexikon der Informatik","Duden-Verlag",2012,"Informatik"));
       liste.add(new Buch("Landschaften Deutschlands","Tegeler, Jürgen",2020));
       liste.add(new Roman("Der Herr der Ringe","J.R. Tolkien","Fantasy", 1964));
       liste.add(new Sachbuch("Biochemie der Pflanze","Nultsch, Karl",1987,"Biochemie"));
    }

    public void zeige()
    {
       for (Buch b: liste)
       {
          b.zeige();
       }
    }
}
```

In der Methode `erzeuge()` werden der Liste ein **Buch**-, ein **Roman**- und drei **Sachbuch**-Objekte zugefügt.

Die `zeige()`-Methode geht die Liste Element für Element durch und ruft dabei für jedes Objekt die *passende* `zeige()`-Methode auf.

Das erste Listenelement ist ein Objekt der Klasse **Sachbuch** (streng genommen: Eine *Referenz* auf ein Objekt der Klasse **Sachbuch**). Die Schleife in der Methode **Buchliste**.`zeige()` führt dann die Methode **Sachbuch**.`zeige()` aus.

Das dritte Listenelement ist ein Objekt der Klasse **Buch**, entsprechend wird beim Anzeigen **Buch**.`zeige()` ausgeführt.

Beim vierten Listenelement wird **Roman**.`zeige()` ausgeführt und beim fünften Element wieder **Sachbuch**.`zeige()`.

Was bedeutet Polymorphie?

Wörtlich übersetzt bedeutet der Begriff "Polymorphie" so viel wie "Vielgestaltigkeit".

Auf unser Beispiel bezogen versteht man unter diesem Begriff Folgendes:

Eine Referenz der Oberklasse **Buch** kann zur Laufzeit auch auf Objekte der Unterklassen **Roman** und **Sachbuch** verweisen. Bei Methodenaufrufen wird dann automatisch die passende überschreibende Methode des tatsächlichen Objekts ausgeführt.

Das ArrayList-Objekt `liste` in der Klasse **Buchliste** ist folgendermaßen definiert:

```
private ArrayList<Buch> liste;
```

Das heißt, in `liste` dürfen nur Objekte gespeichert werden, die entweder vom Typ **Buch** sind oder einer Unterklasse von **Buch** angehören.

Somit handelt es sich bei dem Objekt `liste` zwar um eine ***heterogene Liste,*** da Objekte verschiedener Klassen in ihr gespeichert werden können. Allerdings dürfen nicht Objekte *beliebiger* Klassen in `liste` gespeichert werden, sondern nur Objekte der Klasse **Buch** und ihrer Unterklassen **Roman** und **Sachbuch**.

Sollten später einmal weitere Unterklassen von **Buch** erstellt werden, so können auch Objekte dieser Unterklassen in der Liste gespeichert werden.

Upcasting beim Einfügen in die Liste

```
liste.add(new Roman(...));
liste.add(new Sachbuch(...));
```

Die Methode `add()` erwartet ein Objekt der Klasse **Buch**. Das Einfügen eines **Sachbuch**-Objekts funktioniert trotzdem, weil ein **Sachbuch** ein **Buch** ist. In der Informatik-Didaktik spricht man auch gern von ***IST-Beziehungen***: Ein Sachbuch IST ein Buch.

Unter dem Begriff ***Upcasting*** versteht man ein Typecasting, also eine Typumwandlung, bei der ein "niedriger" Typ (Unterklasse) in einen "höheren" Typ (Oberklasse) umgewandelt wird. Das übergebene **Sachbuch**-Objekt wird beim Einfügen in die Liste in ein **Buch**-Objekt "hochgestuft" - daher Upcasting.

Polymorphie beim Durchlaufen der Liste

```
for (Buch b : liste)
{
    b.zeige();
}
```

Die Variable `b` ist eigentlich vom Typ **Buch**. Zur Laufzeit kann sie jedoch als Referenz-Variable auf ein **Buch**-Objekt, ein **Roman**-Objekt oder ein **Sachbuch**-Objekt verweisen.

Beim Aufruf `b.zeige()` entscheidet Java zur Laufzeit, *welche* `zeige()`-Methode wirklich gemeint ist.

Die ***Polymorphie*** ermöglicht es uns, *eine einzige* Schleife mit einem einzigen Methodenaufruf zu schreiben, die für jedes Objekt in der Liste die entsprechende `zeige()`-Methode aufruft, basierend auf dem Buch**-Typ.**

Ein Vorteil dieses Verfahrens: Wenn wir weitere Unterklassen von **Buch** anlegen, beispielsweise **Fachbuch** oder **Lexikon**, dann kann die obige for-each-Schleife *unverändert* übernommen werden.

9.2.4 Eine bessere Konsolenausgabe

Der IST-Zustand

So ganz perfekt ist die Konsolenausgabe der Klasse **Buchliste** noch nicht:

IST-Zustand	SOLL-Zustand

`--------------------------------`
`Titel : Einführung in Java`
`Autor : Meier, Otto`
`Jahr  : 2025`
`_____________________________/`
 	
`Fachgebiet  : Informatik`
`--------------------------------`
`Titel : Lexikon der Informatik`
`Autor : Duden-Verlag`
`Jahr  : 2012`
`_____________________________/`
 	
`Fachgebiet  : Informatik`
`--------------------------------`
`Titel : Landschaften Deutschlands`
`Autor : Tegeler, Jürgen`
`Jahr  : 2020`
`_____________________________/`
 	
`--------------------------------`
`Titel : Der Herr der Ringe`
`Autor : J.R. Tolkien`
`Jahr  : 1964`
`_____________________________/`
 	
`Genre  : Fantasy`
`--------------------------------`
`Titel : Biochemie der Pflanze`
`Autor : Nultsch, Karl`
`Jahr  : 1987`
`_____________________________/`
 	
`Fachgebiet  : Biochemie`
	`--------------------------------`
	`Titel : Einführung in Java`
	`Autor : Meier, Otto`
	`Jahr  : 2025`
	`Fachgebiet  : Informatik`
	`--------------------------------`
	
	`--------------------------------`
	`Titel : Lexikon der Informatik`
	`Autor : Duden-Verlag`
	`Jahr  : 2012`
	`Fachgebiet  : Informatik`
	`--------------------------------`
	
	`--------------------------------`
	`Titel : Landschaften Deutschlands`
	`Autor : Tegeler, Jürgen`
	`Jahr  : 2020`
	`--------------------------------`
	
	`--------------------------------`
	`Titel : Der Herr der Ringe`
	`Autor : J.R. Tolkien`
	`Jahr  : 1964`
	`Genre  : Fantasy`
	`--------------------------------`
	
	`--------------------------------`
	`Titel : Biochemie der Pflanze`
	`Autor : Nultsch, Karl`
	`Jahr  : 1987`
	`Fachgebiet  : Biochemie`
	`--------------------------------`

Die `zeige()`-Methoden der Unterklassen rufen zunächst die `zeige()`-Methode der Oberklasse **Buch** auf. Diese schreibt dann die Werte der drei Instanzvariablen `titel`, `autor` und `jahr` in die Konsole, eingerahmt von zwei gestrichelten Linien:

```
--------------------------------
Titel : Einführung in Java
Autor : Meier, Otto
Jahr  : 2025
_____________________________/
```

Erst nachdem `Buch.zeige()` fertig ist, geben die Unterklassen ihr Genre (bei **Roman**) bzw. das Fachgebiet (bei **Sachbuch**) aus. Dadurch steht diese Zusatzinformation unterhalb der abschließenden Linie, was nicht gut aussieht.

Wie kann man diese Ausgabe besser gestalten?

Eine bessere Ausgabe

Die Klasse **Buchliste** müssen wir nicht verändern. Stattdessen erweitern wir die Klassen **Buch**, **Roman** und **Sachbuch** so, dass auch die Zusatzdaten *innerhalb* des Rahmens ausgegeben werden.

Die Idee: Die Oberklasse **Buch** erhält eine zusätzliche Methode zeigeDaten(). Diese Methode ist in der Oberklasse zunächst leer, wird aber von der Methode zeige() an der passenden Stelle aufgerufen. Die Unterklassen überschreiben anschließend zeigeDaten() und geben dort ihre speziellen Zusatzinformationen aus.

Überarbeiteter Code in der Klasse **Buch**:

```
    public void zeige()
    {
        System.out.println("--------------------------------");
        System.out.println("Titel : " + titel);
        System.out.println("Autor : " + autor);
        System.out.println("Jahr  : " + jahr);
        zeigeDaten();
        System.out.println("--------------------------------");
        System.out.println();
    }

    public void zeigeDaten()
    {

    }
```

Die neue Methode zeigeDaten() enthält in der Oberklasse **Buch** keinen Code, sondern dient als ***Platzhalter*** für die Unterklassen. Wichtig ist die ***Position*** des Methodenaufrufs innerhalb der Methode zeige(): Die Methode wird nach der Ausgabe der allgemeinen Daten, aber vor der unteren Trennlinie aufgerufen.

Die Unterklassen *können* zeigeDaten() nun überschreiben und dort ihre eigenen Zusatzinformationen ausgeben. Wird später die Methode zeige() aufgerufen, so werden zuerst die allgemeinen Daten des Buchs ausgegeben. Anschließend wird automatisch die passende zeigeDaten()-Methode der jeweiligen Unterklasse ausgeführt. Danach setzt die Methode zeige() der Oberklasse ihre Arbeit fort und gibt die untere Trennlinie aus. Dann sollte das Ganze wie gewünscht funktionieren. Hier der überarbeitete Quelltext der Klasse **Roman**:

```
public class Roman extends Buch
{
    private String genre;

    public Roman(String titel, String autor, String genre, int jahr)
    {
        super(titel, autor, jahr);
        this.genre = genre;
    }

    @Override
    public void zeigeDaten()
    {
        System.out.println("Genre : " + genre);
    }
}
```

Die Oberklassen-Methode zeige() wird nicht mehr überschrieben, stattdessen überschreibt die Klasse **Roman** die **Buch**-Methode zeigeDaten().

Was passiert jetzt zur Laufzeit?

Schauen wir uns die Ausgabemethode von **Buch** noch einmal an:

```
public void zeige()
{
    System.out.println("--------------------------------");
    System.out.println("Titel : " + titel);
    System.out.println("Autor : " + autor);
    System.out.println("Jahr  : " + jahr);
    zeigeDaten();
    System.out.println("_______________________________________/");
    System.out.println();
}
```

Zunächst wird die obere Linie ausgegeben, dann werden die Werte der drei Instanzvariablen angezeigt. Beim Aufruf von zeigeDaten() greift jetzt das Polymorphie-Konzept: Wenn das aktuelle Objekt ein **Roman** ist, wird nicht die leere Methode **Buch**.zeigeDaten() ausgeführt, sondern die überschreibende Methode **Roman**.zeigeDaten():

```
@Override
public void zeigeDaten()
{
    System.out.println("Genre : " + genre);
}
```

Dadurch erscheint das Genre an der richtigen Stelle im Rahmen.

Danach geht es in **Buch**.zeige() weiter, und die untere Linie wird ausgegeben.

So entsteht eine saubere, einheitliche Konsolenausgabe, ohne dass die Klasse **Buchliste** geändert werden muss:

```
--------------------------------
Titel : Einführung in Java
Autor : Meier, Otto
Jahr  : 2025
Fachgebiet  : Informatik
_______________________________________/

--------------------------------
Titel : Lexikon der Informatik
Autor : Duden-Verlag
Jahr  : 2012
Fachgebiet  : Informatik
_______________________________________/

--------------------------------
Titel : Landschaften Deutschlands
Autor : Tegeler, Jürgen
Jahr  : 2020
_______________________________________/

--------------------------------
Titel : Der Herr der Ringe
Autor : J.R. Tolkien
Jahr  : 1964
Genre : Fantasy
_______________________________________/

--------------------------------
Titel : Biochemie der Pflanze
Autor : Nultsch, Karl
Jahr  : 1987
Fachgebiet  : Biochemie
_______________________________________/
```

9.2.5 Zwei weitere Unterklassen

Wir wollen die Klasse **Sachbuch** jetzt in weitere Unterklassen untergliedern, ganz nach dem Prinzip der ***Spezialisierung***. Um die Sache übersichtlich zu halten, begnügen wir uns mit zwei Unterklassen von **Sachbuch**: **Lehrbuch** und **Lexikon**. Als zusätzliche Eigenschaft der Klasse Lehrbuch wählen wir das Anforderungsniveau: Anfänger, Fortgeschrittene, Experten. Die zusätzliche Eigenschaft der Klasse Lexikon soll die Zahl der Stichworte sein.

Die Klasse Lexikon

Beginnen wir mit der neuen Unterklasse **Lexikon**.

```
public class Lexikon extends Sachbuch
{
    int artikelzahl;

    public Lexikon
      (String titel, String autor, String gebiet, int artikelzahl, int jahr)
    {
        super(titel,autor,gebiet,jahr);
        setArtikelzahl(artikelzahl);
    }

    public void setArtikelzahl(int zahl)
    {
        if (zahl < 1000)
            throw new IllegalArgumentException
                ("Zu geringe Anzahl an Artikeln: " + zahl);
        if (zahl > 100000)
            throw new IllegalArgumentException
                ("Zu große Anzahl an Artikeln: " + zahl);
        artikelzahl = zahl;
    }

    public int getArtikelzahl()
    {
       return artikelzahl;
    }

    public void zeigeSpezifischeDaten()
    {
        super.zeigeSpezifischeDaten();
        System.out.println("Artikel : " + getArtikelzahl());
    }
}
```

Wir erweitern dann die Methode `erzeuge()` der Klasse **Buchliste** um eine Zeile:

```
public void erzeuge()
{
    liste.add(new Buch("Einführung in Java","Meier, Otto",2025));
    liste.add(new Sachbuch("Lexikon der Informatik","Duden-Verlag","Informatik",2012));
    liste.add(new Roman("Der Herr der Ringe","J.R. Tolkien","Fantasy", 1964));
    liste.add(new Buch("Prinzipien der OOP","Müller, Jonathan",2024));
    liste.add(new Lexikon("Römpps Chemielexikon","Römpp, Justus","Chemie",7500,1989));
}
```

Die Konsolenausgabe sieht aus wie erwartet (hier nur die beiden letzten Bücher):

```
------------------------------------------------
Titel       : Biochemie der Pflanze
Autor       : Nultsch, Karl
Jahr        : 1987
Fachgebiet  : Biochemie
________________________________________________/

------------------------------------------------
Titel       : Römpp Lexikon der Chemie
Autor       : Römpp et al.
Jahr        : 1989
Fachgebiet  : Chemie
Artikelzahl : 6500
________________________________________________/
```

Die Methode zeigeDaten() der Unterklasse **Lexikon** überschreibt direkt zeigeDaten() der Klasse **Sachbuch** und indirekt auch zeigeDaten() der Oberklasse **Buch**. **Sachbuch** hat zeigeDaten() nämlich bereits von **Buch** geerbt und hat diese Methode selbst überschrieben.

Polymorphie: Wenn bei einem **Lexikon**-Objekt die Methode zeige() aus **Buch** ausgeführt wird, ruft diese intern zeigeDaten() auf. Wegen der Polymorphie wird dann die *speziellste* Version ausgeführt, also nicht **Buch**.zeigeDaten(), auch nicht **Sachbuch**.zeigeDaten(), sondern **Lexikon**.zeigeDaten().

Die Klasse Lehrbuch mit einer Enumeration

Bei der Klasse **Lehrbuch** führen wir als zusätzliches Attribut das Anforderungsniveau ein. Dabei lernen wir zugleich ein neues Java-Konstrukt kennen: die ***Enumeration*** beziehungsweise den sogenannten ***Aufzählungstyp***.

Hier zunächst die neue Unterklasse **Lehrbuch**:

```
public class Lehrbuch extends Sachbuch
{
    private Niveau niveau;

    public Lehrbuch
        (String titel, String autor, int jahr, String gebiet, Niveau niveau)
    {
        super(titel, autor, gebiet, jahr);
        this.niveau = niveau;
    }

    @Override
    public void zeigeSpezifischeDaten()
    {
        System.out.printf("%-12s: %-40s%n", "Fachgebiet", fachgebiet);
        System.out.printf("%-12s: %-40s%n", "Niveau", niveau);
    }
}
```

Das Anforderungsniveau wird hier nicht als String verwaltet, sondern als Aufzählungstyp bzw. Enumeration.

Enumerations

Schauen wir uns die Enumeration **Niveau** näher an:

```
public enum Niveau
{
    ANFAENGER,
    FORTGESCHRITTENE,
    EXPERTEN
}
```

Eine Enumeration ist eine spezieller Klassentyp, der durch das Schlüsselwort `enum` eingeleitet wird. Ein solcher Aufzählungstyp benötigt meistens weder Konstruktor noch Methoden, sondern nur die Werte, die möglich sein sollen. In der Regel schreibt man diese Werte - ähnlich wie wie Java-Konstanten - in GROSSBUCHSTABEN.

Eine Enumeration verhält sich in gewisser Weise wie eine Klasse mit festen Konstanten. Man kann daher auf die Werte zugreifen, ohne zuvor ein Objekt der Enumeration erzeugen zu müssen. Betrachten wir dazu die entsprechende Stelle der Klasse **Buchliste**:

```
liste.add(new Sachbuch("Biochemie der Pflanze","Nultsch, Karl",1987,"Biochemie"));
liste.add(new Lexikon("Römpp Lexikon der Chemie","Römpp et al.", 1989, "Chemie", 6500));
liste.add(new Lehrbuch("Java lernen","Müller, Anna", 2024,"Informatik",Niveau.ANFAENGER));
```

Wir können hier einfach Niveau.ANFAENGER schreiben, ähnlich wie bei einer Konstanten (zum Beispiel Math.PI oder Color.BLACK).

Vorteile von Aufzählungstypen

Enumerationen eignen sich immer dann, wenn ein Attribut nur eine kleine, fest vorgegebene Menge von Werten annehmen darf. Statt beliebige Strings zu verwenden, können wir dadurch einen eigenen Datentyp mit genau den erlaubten Werten definieren. Bei der Klasse **Lehrbuch** verhindern wir so, dass Objekte mit einem unsinnigen Anforderungsniveau wie beispielsweise "Dummkopf" erzeugt werden können.

Fassen wir die Vorteile, die Enumerationen bieten, einmal kurz und übersichtlich zusammen:

- Falsche oder unsinnige Werte werden bereits beim Kompilieren verhindert.
- Der Quelltext wird besser lesbar und verständlicher.
- Die Entwicklungsumgebung kann automatisch gültige Werte vorschlagen.
- Tippfehler werden vermieden.
- Änderungen an den erlaubten Werten lassen sich zentral an einer Stelle durchführen.

Hätten wir in der Klasse **Lehrbuch** statt der Enumeration **Niveau** einfach eine Instanzvariable `niveau` von Typ **String** gewählt, wären auch fehlerhafte Eingaben wie "Anfänger", "anfaenger" oder unsinnige Angaben wie "anhänger" oder "dummy" möglich.

Enumerations als innere Klassen

Man hätte die Aufzählung des Niveaus auch als innere Klasse von **Lehrbuch** implementieren können:

```
public class Lehrbuch extends Sachbuch
{
    public enum Niveau
    {
        ANFAENGER,
        FORTGESCHRITTENE,
        EXPERTEN
    }

    private Niveau niveau;

    public Lehrbuch(String titel, String autor, int jahr, String gebiet, Niveau niveau)
    {
        super(titel, autor, jahr, gebiet);
        this.niveau = niveau;
    }

    @Override
    public void zeigeDaten()
    {
        System.out.printf("%-12s: %-40s%n", "Fachgebiet", fachgebiet);
        System.out.printf("%-12s: %-40s%n", "Niveau", niveau);
    }
}
```

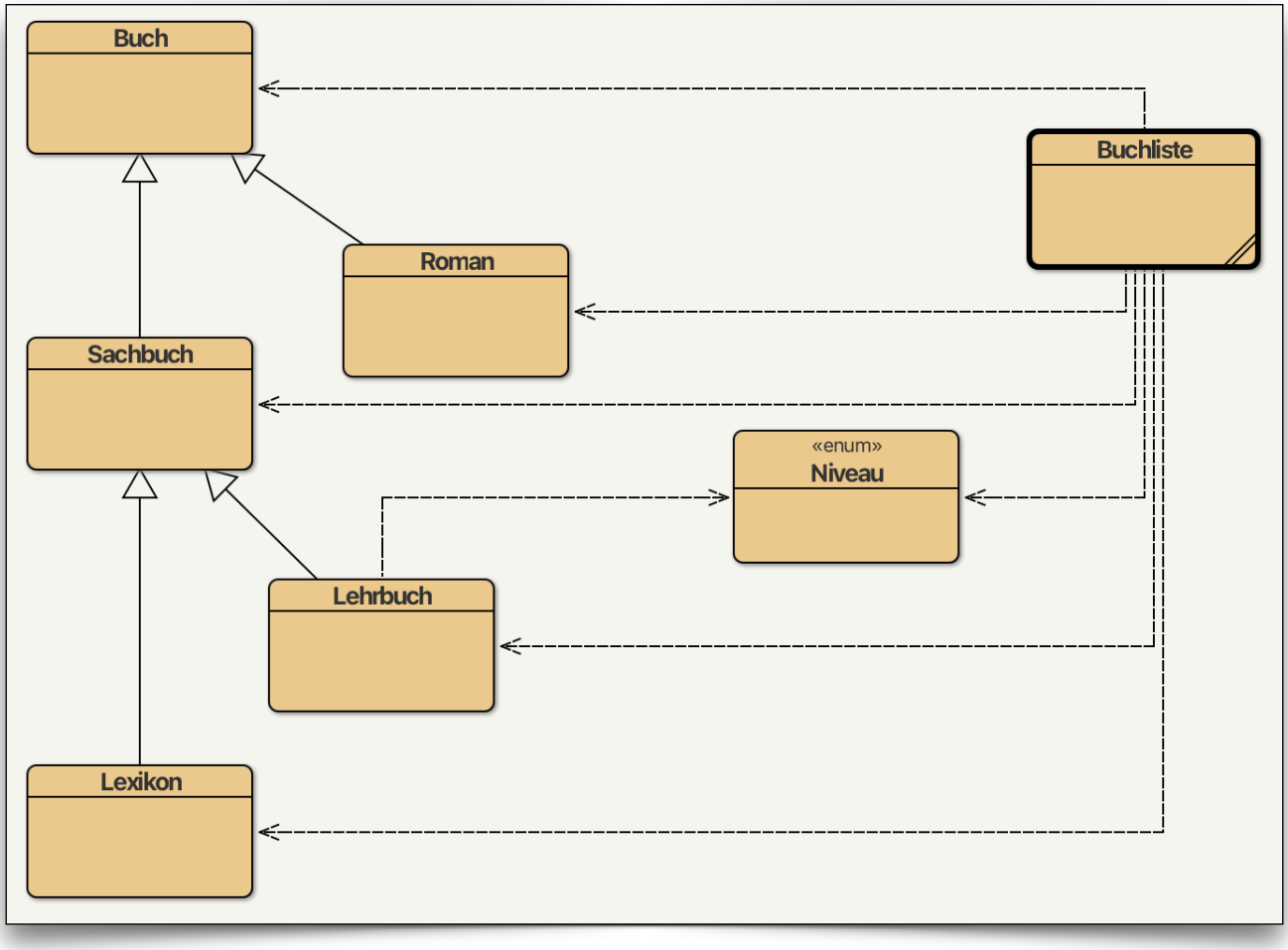
Allerdings wäre dann der Zugriff auf die Werte von außen etwas komplizierter, wie das folgende Beispiel zeigt:

```
Lehrbuch lb = new Lehrbuch
   ("Java lernen","Müller, Anna",2024,"Informatik",Lehrbuch.Niveau.ANFAENGER);
```

Man gibt also zunächst die äußere Klasse **Lehrbuch** an, welche die innere Klasse enthält, dann die innere Klasse **Niveau**, und schließlich den Wert `ANFAENGER`. Möglich ist das, weil die innere Klasse als `public` definiert wurde, nur so ist ein Zugriff von Klassen außerhalb von **Lehrbuch** möglich.

9.2.6 Klassenhierarchie und Klassendiagramm

In der Entwicklungsumgebung BlueJ wird die Klassenhierarchie, die wir bisher erzeugt haben, sehr übersichtlich dargestellt:



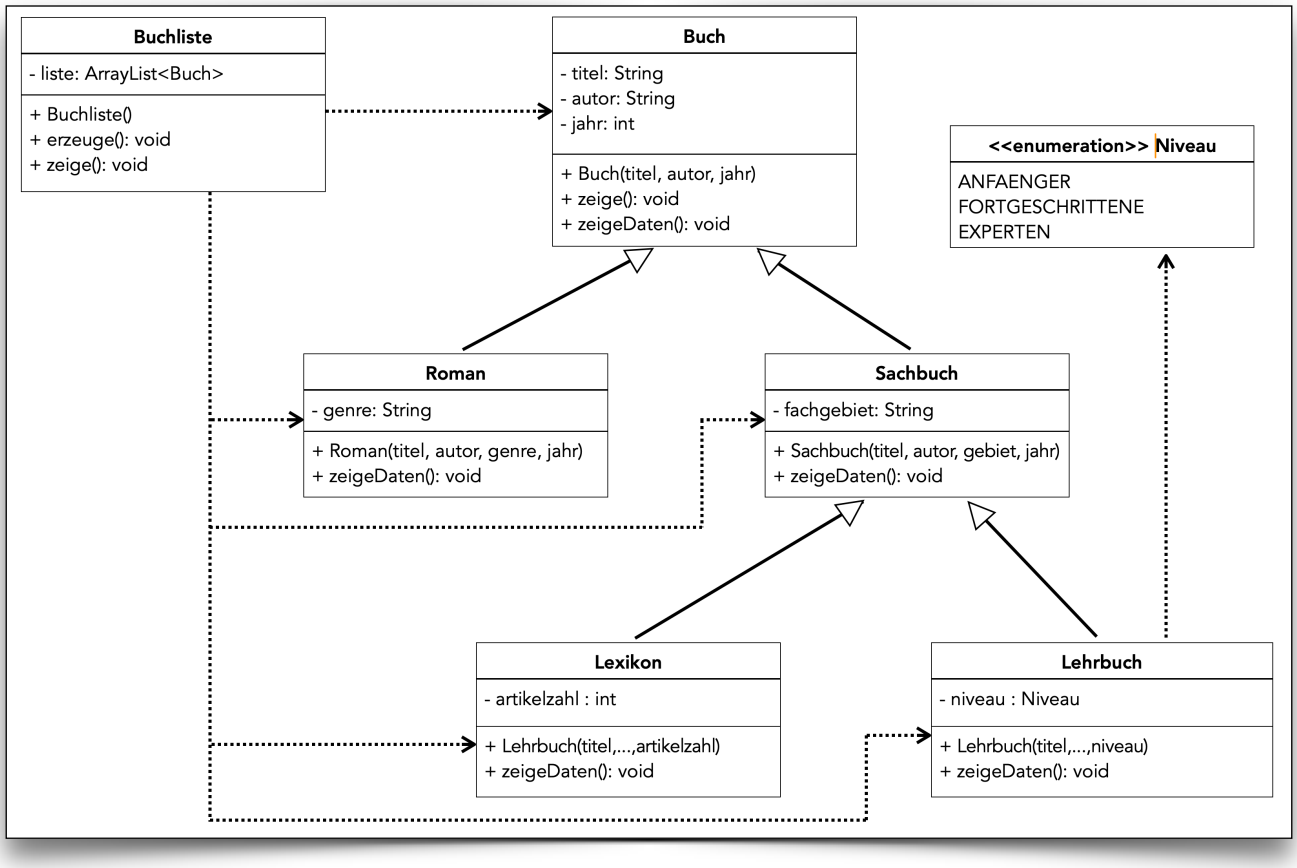
Die ***Vererbungs-Beziehungen*** werden hier durch spezielle Pfeile dargestellt, die von der Unterklasse auf die jeweilige Oberklasse verweisen.

Man erkennt sofort: **Roman** und **Sachbuch** sind Unterklassen von **Buch** (erkennbar an den typischen Pfeilen, die in einem weißen Dreieck enden). **Lexikon** und **Lehrbuch** sind Unterklassen von **Sachbuch**. In der gymnasialen Oberstufe werden solche Beziehungen gern als ***IST-Beziehungen*** bezeichnet.

Weiterhin erkennt man, dass die Klasse **Buchliste** Objekte der Klassen **Buch**, **Roman**, **Sachbuch**, **Lehrbuch** und **Lexikon** sowie der Enumeration **Niveau** besitzt bzw. Zugriff darauf hat (***HAT-Beziehung*** nennt man solche Beziehungen in der Oberstufe).

UML-Klassendiagramme

Einen bessern Überblick bietet folgendes ***UML-Klassendiagramm***:



Allgemeines zu UML-Klassendiagrammen

Eine Klasse wird in einem solchen Diagramm mit einem dreigeteilten Kasten dargestellt. Im oberen Teil steht der **Klassenname**, im mittleren Teil sind die wichtigsten **Instanzvariablen** aufgeführt, und im unteren Teil die wichtigsten **Methoden**.

In einem UML-Diagramm muss man nicht *alle* Instanzvariablen und Methoden aufführen, sondern nur jeweils wichtigen.

Ein Plus-Zeichen vor einer Instanzvariablen oder einer Methode heißt immer `public`, ein Minuszeichen `private`. Die Datentypen stehen - abgetrennt durch einen Doppelpunkt - *hinter* dem Bezeichner der Variable oder Methode.

Die Parameter von Methoden kann man ganz weglassen, nur die Namen hinschreiben oder die Namen mit den Datentypen aufführen; man hat hier also sehr viele Freiheiten.

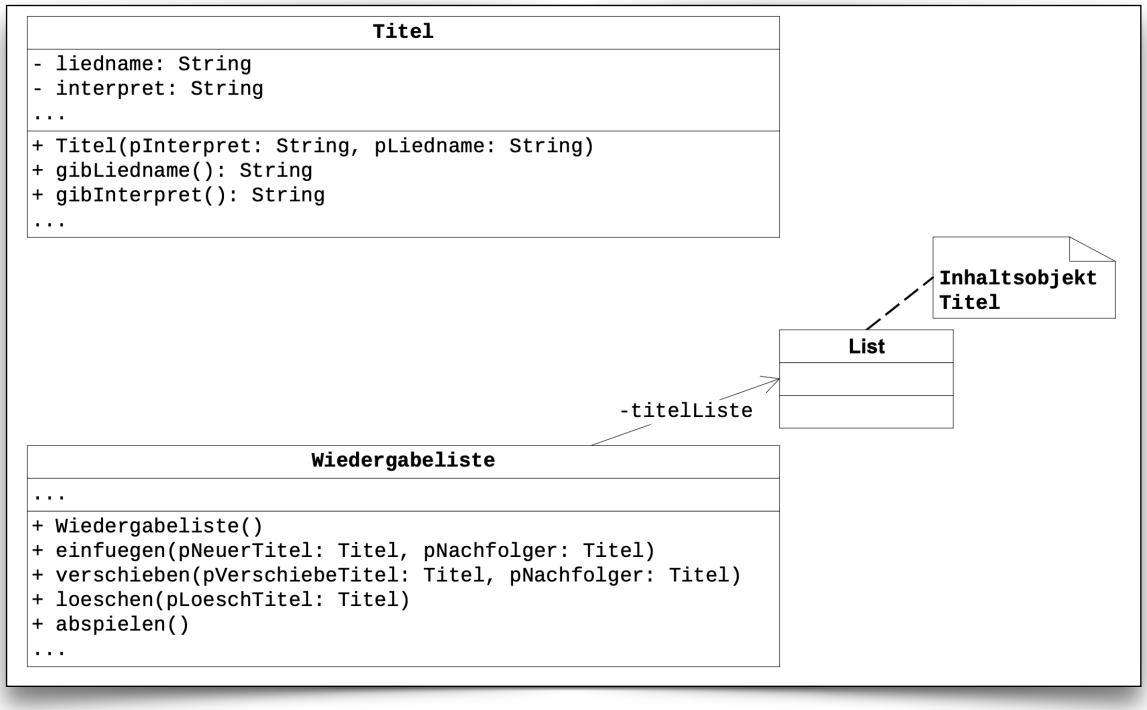
Die durchgezogenen Pfeile mit den weißen Dreiecken am Ende symbolisieren die Vererbungsbeziehungen (IST-Beziehungen), während die gestrichelten Pfeile für Assoziationen stehen (HAT-Beziehungen).

Beispiele für UML-Klassendiagramme

Die folgenden Beispiele für UML-Klassendiagramm stammen aus dem Zentralabitur Informatik des Landes NRW. Mithilfe dieser Beispiele können Sie Ihre Kenntnisse zu diesem Thema vertiefen.

Beispiel 1 (Abitur 2012)

Aus der Aufgabe IF GK HT 1 von 2012



Hier sehen wir ein auf den ersten Blick recht einfaches UML-Klassendiagramm, das aber bei näherer Betrachtung doch etwas komplexer erscheint.

Klasse Titel

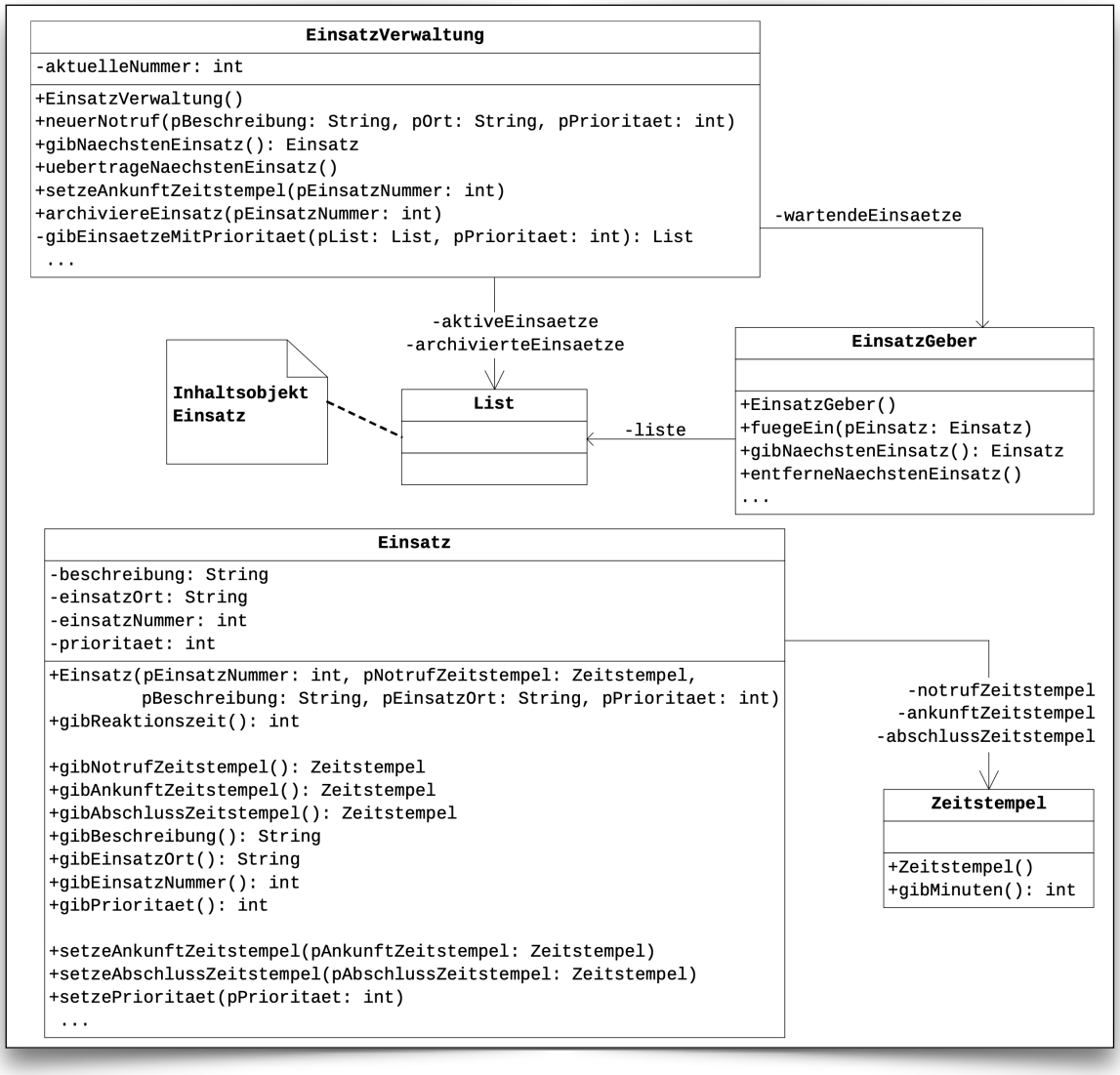
Eine Klasse **Titel** hat mehrere Instanzvariablen, von denen offensichtlich `liedname` und `interpret`, beide vom Typ **String**, die wichtigsten für die Aufgabenstellung sind. Weitere Instanzvariablen sind möglicherweise vorhanden, werden in diesem Diagramm aber vernachlässigt.

Die Klasse **Titel** besitzt einen Konstruktor, dem der Liedname und der Interpret übergeben werden. Außerdem sehen wir zwei Getter-Methoden `gibLiedname()` und `gibInterpret()`. Weitere Methoden sind durch ... an gedeutet.

Klasse Wiedergabeliste

Die Klasse besitzt eine Instanzvariable `titelliste`, die aber nicht in dem zweiten Abschnitt des Klassenkastens aufgeführt ist. Diese `titelliste` ist ein Objekt einer Klasse **List**, bei der es sich offensichtlich um eine Sammlungsklasse wie **ArrayList** oder **LinkedList** handelt. Die Elemente dieser Sammlungsklasse - hier als ***Inhaltsobjekte*** bezeichnet - sind Verweise auf Objekte der Klasse **Titel**.

Beispiel 2 (Abitur 2015)



Dieses UML-Diagramm aus dem Jahre 2015 sieht schon komplexer aus.

Klasse Einsatz

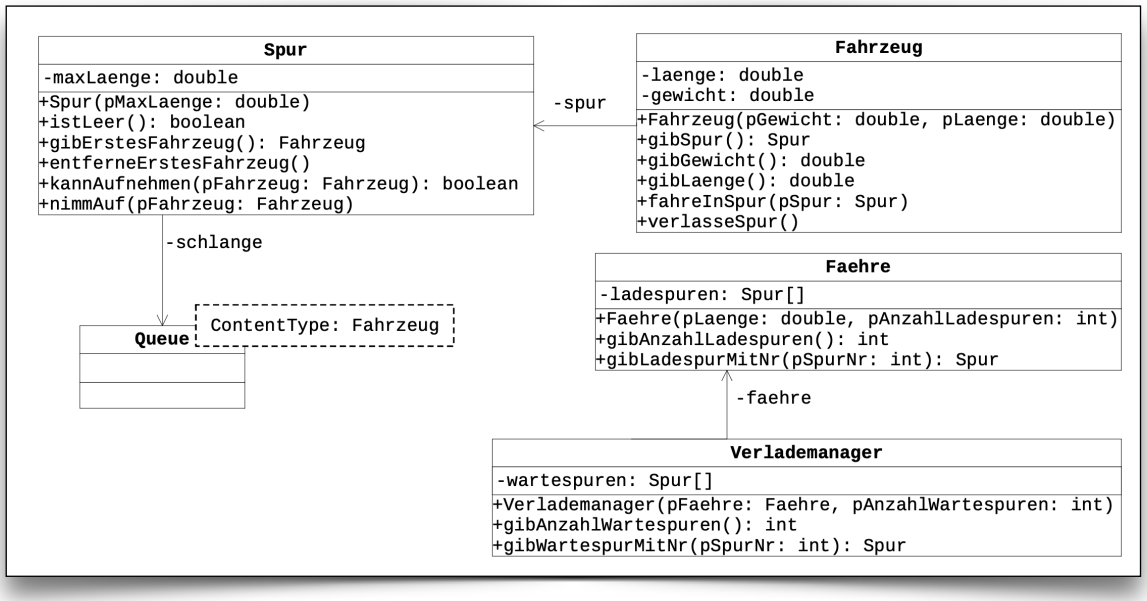
Diese Klasse ist die elementare Klasse der Einsatzverwaltung einer Polizeistation. Auf die Instanzvariablen sowie die vielen Methoden gehen wir jetzt nicht im Einzelnen ein. Nicht direkt im Klassenkasten stehen hier drei private Instanzvariablen `notrufZeitstempel`, `ankunftZeitstempel` und `abschlussZeitstempel`, bei denen es sich um Objekte der Klasse **Zeitstempel** handelt.

Klasse Einsatzverwaltung

Auch hier vernachlässigen wir die Instanzvariablen und Methoden im Klassenkasten und schauen uns lieber die Instanzvariablen `aktivierteEinsaetze` und `archivierteEinsaetze` an, die nicht direkt im Klassenkasten vermerkt sind. Bei beiden Instanzvariable handelt es sich um Objekte einer Sammlungsklasse **List**, deren Elemente Verweise auf Objekte der Klasse **Einsatz** sind.

Eine dritte Instanzvariable von Einsatzverwaltung heißt `wartendeEinsaetze` und ist ein Objekt der Klasse **EinsatzGeber**. Eine Instanzvariable dieser Klasse heißt `liste` und ist ebenfalls ein Objekt der Klasse **List**.

Beispiel 3 (2017)



Dieses dritte UML-Diagramm aus einer Abituraufgabe des Jahres 2017 beschreibt ein kleines Programm zur Verwaltung einer Fähre mit Fahrzeugen und Wartespuren.

Die zentrale Klasse ist **Fahrzeug**. Ein Fahrzeug besitzt die beiden Instanzvariable `laenge` und `gewicht`. Außerdem kann ein Fahrzeug in eine Spur einfahren oder diese wieder verlassen. Dazu besitzt die Klasse unter anderem die Methoden fahreInSpur() und verlasseSpur(). Über die Methode gibSpur() kann abgefragt werden, in welcher Spur sich das Fahrzeug gerade befindet.

Die Klasse **Spur** verwaltet mehrere Fahrzeuge. Intern verwendet sie dazu ein Objekt der Klasse **Queue**, also eine Warteschlange. Die Elemente dieser Warteschlange sind Referenzen auf Objekte der Klasse **Fahrzeug**. Eine Spur besitzt außerdem eine maximale Länge, die im Attribut `maxLaenge` gespeichert wird.

Mit den Methoden gibErstesFahrzeug() und entferneErstesFahrzeug() kann auf das erste Fahrzeug der Warteschlange zugegriffen werden. Die Methode kannAufnehmen() überprüft, ob noch genügend Platz für ein weiteres Fahrzeug vorhanden ist. Mit nimmAuf() wird ein Fahrzeug schließlich in die Spur eingefügt.

Die Klasse **Faehre** besitzt mehrere Ladespuren, die in einem Array vom Typ `Spur[]` gespeichert werden. Über die Methode gibLadespurMitNr() kann auf eine bestimmte Spur zugegriffen werden.

Die Klasse **Verlademanager** schließlich verwaltet die Wartespuren vor der Fähre. Auch diese Wartespuren werden in einem Array vom Typ `Spur[]` gespeichert. Zusätzlich besitzt der Verlademanager eine Instanzvariable, die ein Objekt der Klasse **Faehre** ist.

Diese drei Beispiele aus dem NRW-Zentralabitur sollten Ihnen einen Einblick in die Möglichkeiten von UML-Klassendiagrammen geben.

9.3 Noch ein paar Kleinigkeiten

9.3.1 Das Schlüsselwort final

Man kann theoretisch von jeder existierenden Java-Klasse Unterklassen anlegen. Manchmal will man das aber aus bestimmten Gründen verhindern. In diesem Fall gibt es einen einfachen Trick, wie man das erreichen kann: Wir geben der Klasse bei der Deklaration das Attribut `final`.

Beispiel:

```
public final class Lexikon extends Sachbuch
```

Dann versuchen wir eine Unterklasse **Chemielexikon** anzulegen:

```
public class Chemielexikon extends Lexikon
{
    public Chemielexikon(String titel, String autor, int jahr)
    {
       super(titel, autor, "Chemie", jahr);
    }
}
```

Der Compiler verweigert die Übersetzung mit der Fehlermeldung:

```
"Erben aus finalem Lexikon-Element nicht möglich".
```

Wann ist das sinnvoll?

Wenn eine Klasse, von der Objekte erzeugt werden können, bestimmte Regeln enthält, die auf keinen Fall geändert werden dürfen, dann kann Vererbung zum Problem werden: Man könnte eine Unterklasse anlegen, die diese in Methoden festgelegten Regeln einfach durch eigene Methoden mit gleicher Signatur und eigenen Regeln überschreibt.

Stellen Sie sich eine Klasse **Konto** vor, bei der die Methode `abheben()` prüft, ob der gewünschte Betrag größer ist als das Guthaben. Ein Entwickler könnte nun eine Unterklasse **WunderKonto** anlegen, in der `abheben()` überschrieben wird, sodass beliebig hohe Beträge abgehoben werden können - vielleicht sogar von einem anderen Konto.

Durch das Schlüsselwort `final` in der Deklaration von **Konto** wird ein solches Vorgehen verhindert, weil dann keine Unterklassen mehr definiert werden können.

Finale Methoden

Steht das Schlüsselwort `final` vor einer Methode, so kann diese Methode von Unterklassen nicht mehr überschrieben werden.

Finale Instanzvariablen

Wenn wir eine Instanzvariable mit `final` deklarieren, kann der einmal zugewiesene Wert nicht mehr verändert werden. Daher nutzt man das Schlüsselwort final beispielsweise, um ***Konstanten*** zu definieren.

9.3.2 Vererben von Konstruktoren

Konstruktoren werden in Java nicht vererbt. Das bedeutet: Auch wenn eine Oberklasse mehrere selbst definierte Konstruktoren besitzt, stehen diese der abgeleiteten Unterklasse nicht automatisch zur Verfügung. Stattdessen müssen alle benötigten Konstruktoren in der Unterklasse neu definiert werden – selbst dann, wenn sie lediglich den Konstruktor der Oberklasse mittels `super()` aufrufen.

"Konstruktoren werden nicht vererbt. Sie müssen alle benötigten Konstruktoren in einer abgeleiteten Klasse neu definieren, selbst wenn sie nur aus einem Aufruf des Superklassenkonstruktors bestehen." (S. Dörn, Java lernen in abgeschlossenen Lerneinheiten)

Beispiel Konto/JugendKonto

Um dieses Prinzip zu veranschaulichen, betrachten wir eine Klasse **Konto** mit zwei Konstruktoren:

```
public class Konto
{
    private String inhaber;
    private int guthaben; // in Cent

    public Konto(String inhaber)
    {
        this(inhaber, 0);
    }

    public Konto(String inhaber, int startGuthaben)
    {
        if (inhaber == null)
            throw new IllegalArgumentException("Inhaber fehlt.");
        if (startGuthaben < 0)
            throw new IllegalArgumentException(
                  "Startguthaben darf nicht negativ sein.");

        this.inhaber = inhaber;
        guthaben = startGuthaben;
    }
}
```

Das ist eine typische Java-Klasse mit zwei Konstruktoren:

1. einem Hauptkonstruktor mit zwei Parametern
2. einem sogenannten Komfort-Konstruktor, der lediglich den Hauptkonstruktor mit dem Startguthaben 0 aufruft.

Wir legen nun eine Unterklasse **JugendKonto** mit einem zusätzlichen Attribut `limit` an:

```
public class JugendKonto extends Konto
{
    private int limit; // in Cent

    public JugendKonto(String inhaber)
    {
        super(inhaber);          // ruft Konto(String) auf
        limit = 2000;            // z.B. 20,00 Euro
    }

    public JugendKonto(String inhaber, int startGuthaben)
    {
        super(inhaber, startGuthaben);  // ruft Konto(String, int) auf
        limitCent = 2000;
    }
}
```

Beide Konstruktoren der Unterklasse beginnen mit einem super(...)-Aufruf, der den jeweils passenden Konstruktor der Oberklasse anspricht. Das ist nicht nur guter Stil, sondern in Java zwingend erforderlich: Der super(...)-Aufruf muss, sofern er explizit (also vom Entwickler) angegeben wird, stets die **erste Anweisung** im Konstruktorrumpf sein.

Erst dann wird das klasseneigene Attribut `limit` initialisiert - ein Schritt, den die Oberklasse naturgemäß nicht übernehmen kann, da sie von `limit` keine Kenntnis hat.

Was würde passieren, wenn die Unterklasse Jugendkonto keine eigenen Konstruktoren hätte?

```
public class JugendKonto extends Konto
{
    private int limit = 2000; // in Cent
}
```

Der Compiler würde so vorgehen:

1. Er stellt fest, dass die Unterklasse **Jugendkonto** keinen Konstruktor besitzt.

2. Er erzeugt daraufhin automatisch einen parameterlosen Standardkonstruktor:

```
public JugendKonto()// parameterloser Standardkonstruktor
{
    super();
}
```

3. Dieser ***implizit*** (also automatisch und für den Entwickler nicht sichtbar) eingefügte super()-Aufruf setzt voraus, dass die Oberklasse ebenfalls einen parameterlosen Konstruktor besitzt.

4. Die Oberklasse **Konto** besitzt jedoch keinen parameterlosen Konstruktor, da bereits zwei eigene Konstruktoren mit Parametern definiert wurden. Sobald eine Klasse mindestens einen Konstruktor ***explizit*** definiert (also durch den Entwickler erstellt), erzeugt der Compiler keinen parameterlosen Standardkonstruktor mehr.

5. Der Aufruf super() kann daher nicht aufgelöst werden, und so meldet der Compiler einen Fehler. Das Programm lässt sich nicht übersetzen.

> Sobald eine Oberklasse **keinen parameterlosen Konstruktor** besitzt, muss jede Unterklasse mindestens einen eigenen Konstruktor definieren, der mit super(...) einen der vorhandenen Oberklassen-Konstruktoren explizit aufruft. Andernfalls verweigert der Compiler die Übersetzung.