Java Uni ab Woche 7

January 8, 2024

Contents

Datentypen und Objekte

Etwas abstrakt formuliert ist ein Datentyp, ein Menge von Werten, zusammen mit Operationen die auf diese Werten definiert sind. Jeder Wert in Java ist also eine Instanz eines Bestimmten Datentyps.

Operationen sind zum beispiel bei **int** die addition oder bei *arrays* die [] um auf elemte zu zugreifen.

Um eine Instanz eines Datentyps zu erstellen benutzen wir das Schlüsselwort new.

Listing 1: Erzeugung Instanz

```
// Erzeugen eines Werts des Datentyp Strings
String s = new String("hello world");
```

Operationen auf eine Wert werden imttels Methodenaufrufung ausgeführt

Listing 2: Erzeugung Instanz

```
// Anwenden der Operation lenght auf dem Wert "hello world"
s.lenght();
```

In der Objektorientierten Programmierung werden Instanzen oder Werte von Datentypen **Objekte** gennant. Die variable **s** aus dem beispiel **referenziert** das Objekt *String*.
Die Primitiven Datentypen (int, double, cahr, etc.) haben eine Wrappenklasse um sie auch wie andere Objekte behandeln zu können.

Primitiver Datentyp	Wrapper
int	Integer
double	Double
boolean	Boolean
char	Character
float	Floar
short	Short
long	Long
byte	Byte

Objektorientierung

Funktionen verwenden wir um Sequenzen von Anwesungen zu abstrahieren. Ähnlich dazu werden wir nun Klassen verwenden um zusammengehörige Daten zu einer grösseren Einheit zusammenzufassen. Werte dieser Klassen (also Instanzen der Datentypen) werden Objekte genannt. Diese Art der Datenorganisation mittels Klassen führt zu einer Art der Softwareentwicklung, die sich Objektorientierte Programmierung nennt.

Klassen

Klassen sind ein Konstrukt mit dem wir Datentypen definieren können. Dafür müssen wir die Werte, welche repräsentiert weden, definieren und Operationen welche auf diesen Werten ausgeführt werden können.

Definition der Klasse

Um eine Klasse zu definieren müssen wir das Schlüsselwort **class** benutzen. Ein beispiel.

Listing 3: Klasse definieren

```
class Point {

}
```

Felder der Klassen

Die Daten die unser Datentyp repräsentiert nennen wir **Felder** oder *Attribute*. Diese definieren wir einfach inerhalt der Klassenrumpfs. In diesem beispiel hat ein Punkt x und y Koordinate, welche wir verwenden und repräsentieren wollen.

Listing 4: Felder

```
class Point {
    double x;
    double y;
}
```

Erzeugen von Objekten

Im Folgenden beispiel erzeugen wir zwei Instanzen des Datentyps **Point** und weisen diese den Vatiablen *point1* un d*point2* zu.

Listing 5: Erzeugen von Objekten

```
Point point1 = new Point();
Point point2 = new Point();
```

Setzen und Lesen der Werte der Felder eines Objekts

Wir wollen point1 die Koordinaten $(0, 0)$ und point2 $(1, 2)$ zuweisen.

Listing 6: zuweisen von Werten auf Felder

```
point1.x = 0;
point1.y = 0;

point2.x = 1;
```

```
point2.y = 2;
```

Listing 7: ausgeben der Werte

```
System.out.println("point1, x-koordinate:" + point1.x);
//...
```

Methoden

Der zweite Teil einer Definition eines Datentyps sind die Operationen, die wir darauf ausführen können. Diese werden über Methoden definiert.
Methode um Punkt um **dx** und **dy** zu verschieben

Listing 8: methode

```
class Point {
        double x;
        double y;

    public void translate(double dx, double dy) {
        x = x + dx;
        y = y + dy;
    }
}
```

Die methode nimmt als Paramater zwei Gleitkommazahlen entgegen und addiert sie zu den bestehenden Variablen hinzu das kann sie weil sie auf diese Zugreifen kann.

Listing 9: ausführen methode

```
p1.translate(2, 3); // Translates object p1 of type Point
```

Konstruktor

Um Werte einem Objekt direkt beim erstellen zu übergeben nutzen wir eine Konstruktor. Dieser muss gleich heissen wie die Klasse.

Listing 10: Konstruktor

```
class Point {
    double x;
    double y;
```

```
        // Definition des Konstruktors
        Point(double xValue, double yValue) {
            x = xValue;
            y = yValue;
        }
    }
```

Listing 11: nutzen Konstruktor

```
        Point p1 = new Point(0, 0);
```

Klassen sind Referenzdatentypen

Eine Variable eines Referenzdatentyps ist nur eine Adresse des eigentlcihen typs. Heisst wir können nicht einfach zwei Objekte gleich setzen und der eine Typ wrid dem anderen übergene. Um Variabeln der Felder genau anzusprechen könne wird das Schlüsselwort **this** vor diese Variabeln setzen.

Klassenfelder und Klassenmethoden

Java unterscheidet Felder wie auch Methoden ob sich diese auf ein Objekt oder eine Klasse beziehen. Wir sprechen dann jeweils von Objektfelder/Objektmethoden und Klassenfelder/Klassenmethoden. Dabei brauchen wir für Objektfelder/Objektmethoden ein Objekt um auf diese zuzugreifen oder diese aufzurufen. Bei den Klassenfeldern/Klassenmethoden verwenden wir stattdessen die Klasse durch deren Klassennamen.
static vor einer Methode oder Feld, heisst diese können genutzt werden ohne Instanz der Klasse erstellt zu haben.

Objektorientierung 2

In diesem Abschnitt werden wir Konsepte kennenlernen welche uns ermöglichen Hierarchien zu erstellen und wir werden Interfaces kennenlernen, mit welchen wir die Operationen eines Datentyps von der Implementation trennen können. Somit können wir Schnittschtelle zwischen verschiedenen Implementation erstellen.

Interfaces

Ein Interface gibt uns die Möglichkeit die Operationen einer Klasse von der Implementation dieser Operationen zu trennen. Ein Interface spezifiziert einfach eine Menge von Methoden, die ein Datentyp zur Verfügung stellen muss.
Wir zeigen an einem Beispiel den Nutzen der Interfaces. Wir wollen eine Anwendung programmierung welche geometrische Elemente zeichnen können soll. Diese Elemente (Kreise, Dreiecke, Rechtecke etc.) sollen auch manipuliert, vergössert, verkleinert und verschoben werden könne.
Um die Objekte zu skalieren, definieren wir auf jedem Datentypen die Methode **scale(double scaleFactor)**

Listing 12: Scalable

```
interface Scalable {
    void scale(double scaleFactor); // Methodensignatur
}
```

Wir definieren uns ein zweites Interface, welche die Methoden für das Positionieren des Objekts zur Verfügung stellt:

Listing 13: Movable

```
interface Movable {
    void translate(double x, double y);
    void moveToOrigin();
}
```

Wenn wir die geometrischen Objekte implementieren könne wir bestimmen welche Interfaces sie unterstützen. Wir sprechen davon, dass eine Klasse ein Interface implementiert. Tatsächlich wird jedoch nicht das Interface implementiert, sondern die im Interface definierten Methoden.
Im Folgenden finden Sie die Beispieldefinition einer Klasse Circle. Diese implementiert die in den Interfaces Movable und Scalable definierten Methoden.

Listing 14: Circle

```
class Circle implements Movable, Scalable {
    double xPos;
    double yPos;
    double radius;

    public void translate(double x, double y) {
        this.xPos = this.xPos + x;
```

```
        this.yPos = this.yPos + y;
    }

    public void moveToOrigin() {
        this.xPos = 0;
        this.yPos = 0;
    }

    public void scale(double scaleFactor) {
        this.radius = this.radius * scaleFactor;
    }
}
```

Nach dem Schlüsselwort **implements** werden die Interfaces angegeben die implementieren werden sollen. Die Methoden müssen im Klassenrumpf definiert und implementiert werde, ebenso müssen sie **public** sein.
Nicht alle Elemente in unserer Anwendung müssen skalierbar sein. So zum beispiel der Datentyp **Point** welcher nur **Movable** implementieren muss.

Listing 15: Point

```
        class Point implements Movable {
            double xPos;
            double yPos;

            public void translate(double x, double y) {
                this.xPos = this.xPos + x;
                this.yPos = this.yPos + y;
            }

            public void moveToOrigin() {
                this.xPos = 0;
                this.yPos = 0;
            }

        }
```

Die Klassen Point und Circle haben nun also beide das Interface Moveable implementiert. Davon können wir in unserer Anwendung Gebrauch machen. Wir können Methoden schreiben, welche für ein Objekt nur ein Interface voraussetzen und benutzen. Es müssen dann lediglich die in

einem spezifizierten Interface definierten Methoden implementiert sein. Es wird jedoch nicht einen konkreter Datentyp verlangt.
Wir können also eine Methode schreiben ohne genau angeben zu müssen von Welchem Datentyp die Methode verwenden will.

Listing 16: moveAround

```
public static void moveAround(Movable m, double xv, double yv) {
    m.translate(xv, yv);
}
```

Beachten Sie, dass wir hier statt des Datentyps Circle oder Point einfach den Namen des Interface Movable verwendet haben. Dadurch weiss Java, dass die Methode translate zur Verfügung steht.

Listing 17: verschieben Objekte

```
Circle aCircle = new Circle();
Point aPoint = new Point();

moveAround(aCircle, 10, 20);
moveAround(aPoint, 0, 1);
```

Wir können auch direkt Variablen vom Typ des Interfaces definieren.

Listing 18: Interface nutzung

```
Movable movable1 = new Circle();
Movable movable2 = new Point();
```

Vererbung und abstrakte Klassen

Mittels Interfaces könne wir Datentypen erstellen welche gleiche Fähigkeiten haben müssen aber sonst nicht miteinander zu tu haben.
Mit hilfe der Vererbung können wir Hierarchien erstellen. Wir können Konzepte erstellen, welche spezifischere Konzepte andere Konzepte sind. Zum beispiel sind Bannane ein Unterkonzept der Frucht.
Solche Hierarchien könne wir mir Abstrakten Klassen erstellen. Diese werden gleich wie andere Klassen definiert nur mit dem Schlüsselwort **abstract**

Listing 19: abstrakte Klasse

```
abstract class Fruit {
```

```
    String name;
    String color;

    public Fruit(String name, String color) {
        this.name = name;
        this.color = color;
    }

    void eat() { System.out.println("eat"); }
}
```

Abstrakte Klassen können das gleiche wie "normale" Klassen. **Von Abstrakten Klassen könne keine Instanzen erstellt werden.** Mittels abstrakter Klassen wird nur das allgemeine Konzept definiert, welches dann von konkreten, spezialisierten Klassen umgesetzt wird.

Listing 20: Fruit

```
class Banana extends Fruit {

    public Banana() {
        super("Banana", "yellow");
    }
}
```

Das Schlüsselwort **extends** zeigt das die Klasse **Banana** von **Fruit** erbt. Die vererbende Klasse wird auch **Superklasse** gennant. Im Konstruktor der Klasse Banana, müssen wir als erste Anweisung den Konstruktor der Superklasse aufrufen. Dazu verwenden wir das Schlüsselwort **super**. Da Banana von Fruit erbt, müssen alle Eigenschaften, die für das allgemeine Konzept Fruit gelten, auch für die konkrete Spezialisierung Banana gelten. Wir können entsprechend auf Instanzen vom Typ Banana auf alle Methoden und Felder von Fruit zugreifen.

Listing 21: Beispiel Superklasse

```
abstract class Fruit {
    String name;
    String color;

    public Fruit(String name, String color) {
        this.name = name;
        this.color = color;
    }
```

```
    void eat() { System.out.println("eat"); }
}

class Banana extends Fruit {

    public Banana() {
        super("Banana", "yellow");
    }
}

class AbstractClassExperiment {

  public static void main(String[] args) {
    Banana aBanana = new Banana();
    aBanana.eat();
    System.out.println(aBanana.color);
  }
}
```

Genau wie bei Interfaces, ist es auch in abstrakten Klassen möglich, Methoden nur zu deklarieren, nicht aber zu implementieren. Dies macht Sinn, da im abstrakten Konzept ja gewisse Verhalten noch unklar sein können, und nur in den konkreten Klassen wohldefiniert sind. In unserem Frucht-Beispiel könnten wir in der Abstrakten Klasse Fruit eine Methode prepare definieren. Diese soll beschreiben, wie eine Frucht zubereitet werden kann. Wie diese aber umgesetzt werden soll, hängt von der konkreten Frucht ab. Nicht jede Frucht wird gleich zubereitet. In Java können wir dies wie folgt umsetzten:

Listing 22: abstrakte Methoden

```
abstract class Fruit {
    String name;
    String color;

    public Fruit(String name, String color) {
        this.name = name;
        this.color = color;
    }
```

```
    abstract void prepare();
}

class Banana extends Fruit {

    public Banana() {
        super("Banana", "yellow");
    }

    void prepare() {
        System.out.println("peel");
        System.out.println("eat");
    }
}

class Mango extends Fruit {

    public Mango(String color) {
        super("Mango", color);
    }

    void prepare() {
        System.out.println("peel");
        System.out.println("slice");
        System.out.println("eat");
    }
}
```

Polymorphismus und Dynamische Bindung

Bei Polymorphismus handelt sich um das Nutzen von Subklassen ohne diese genau anzusprechen. Zu beispeil wenn die Klasse **Banana** die abstrakte Klasse **Fruit** extends und wir in einer Methode als Parameter **Fruit** angeben aber diese Methode auch eine Instanz von **Banana** entgegen nehmen kann.
Die idee von Dynamischer Bindung ist, dass wenn wir in einem Interface oder einer abstrakten Klasse eine Methode definieren und diese denn auf Subklassen implementieren, wir diese Flexibel einsetzen können. Wenn wir eine Methode schreiben die dann z.b **Fruit** entgegen nimmt und **prepare()** auf **Banana** und **Mango** definiert sind. Könne wir **prepare** auf Fruit anwenden,

obwohl sie dort nicht implementiert ist und dann wird sie so ausgeführt abhengig davon was der Ursprünglischen Methode gegeben wird. Ob eine Instanz von **Mango** oder **Banana**. Hier der Code dazu

Listing 23: Polymorphismus

```
abstract class Fruit {

    abstract void prepare();

}

class Banana extends Fruit {

    void prepare() {
        System.out.println("peel");
        System.out.println("eat");
    }
}

class Mango extends Fruit {

    void prepare() {
        System.out.println("peel");
        System.out.println("cut");
        System.out.println("eat");
    }

}

class FruitExample {

    static void preparationInstructions(Fruit aFruit) {
        aFruit.prepare(); // abhaengig von input in Methode
    }

    public static void main(String[] args) {
        Banana aBanana = new Banana();
        Mango aMango = new Mango();
```

```
        preparationInstructions(aBanana);
    }
}
```

Klassenhierarchie

Java ist in einer Hierarchie aufgebaut. Und ganz oben ist der Datentyp **Object**. Jede neue Klasse erbt von diesem Typ. In dem Datentyp **Object** werden wichtige Methoden definiert welche auf jedem objekt verwendet werden können. Es gitb weitere Hierarchien wie das **Integer** von **Number** erbt.

Downcasting

Wir könne also in eine Variable vom Tyo **Object** eine Instanz vom Typ Integer machen, weil **Object** ganz obe in der Hierarchie ist.

Listing 24: erbt von Object

```
Object anInteger = new Integer(5);
Object aString = new String("Hello world");
```

Umgekehrt geht das aber nicht, da java nicht garantieren kann, das beim zurück welchseln nicht ein andere Wert eingesetzt wurde der nicht eine Subklasse der spezifischen Superklasse ist.

Listing 25: fehler

```
Object anInteger = new Integer(5);
Integer theSameInterger = anInteger; // Fehler
```

Wir könne java aber dazu zwingen, dass es diesen prompt akzeptiert, indem wir das ganze casten.

Listing 26: casting

```
Object anInteger = new Integer(5);
Integer theSameInterger = (Integer) anInteger; // Dies ist gueltig
```

Wenn jetzt aber **anInteger** nicht ein Integer ist gibt es einen Laufzeitfehler. Wir können mit **instanceof** herrausfinden ob ein Object ein Subklasse von einem anderen Object ist. Zum beispiel gibt **anInteger instanceof Integer** true aus.

Equals und toString

Die klasse **Object** definiert wichtige Methoden. Die zwei wichtigsten für diese Zusammenzufassen ist **Equals und toString**.
Die Methode **toString** gibt uns die möglichkeit Objekte als String dar zu stellen. Zum Beispiel ist

Listing 27: toString

```
System.out.println(obj);
```

als **obj.toString** zu verstehen.
In der Klasse **Object** wird die Methode **toString** zwar implementiert aber nur seht allgemein da diese nicht viel über das Objekt weis an welchem die Methode ausgeführt wird. Es wird nur der Name der Klasse und eine interne Referenz ausgegeben. Am beispiel der Point Klasse könne wir das gut sehen.

Listing 28: Point

```
class Point {
    double x;
    double y;
    Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
class ToString {
  public static void main(String[] args) {
    Point aPoint = new Point(3, 4);
    System.out.println(aPoint);
  }
}
```

Ausgabe: Point@6d06d69c
Um eine sinvolle ausgabe zu mache müssen wir **toString** implementiere. Und dabei müssen wir die originale methode toString überschreiben und das machen wir mit dem Schlüsselwort **@Override**.

Listing 29: point

```
class Point {
```

```
    double x;
    double y;
    Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    @Override
    public String toString() {
        return "(" +x + "," +y + ")";
    }
}
```

Die Equals Methode

Die Equals methode hat die Signatur:

Listing 30: Equals

```
public boolean equals(Object obj) ;
```

Die Methode will das *this* objekt mit dem Paramater objekt vergleiche. Das machen wir so:

Listing 31: equals implementiert

```
class Point {
    double x;
    double y;
    Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    @Override
    public boolean equals(Object obj) {

        if (!(obj instanceof Point))
            return false;
        else {
          Point other = (Point) obj;
          return this.x == other.x && this.y == other.y;
        }
    }
```

```
}
```

Pakete, Module und Sichtbarkeit

Um den code besser zu strikturieren und die Wartbarkeit zu gewehrleisten, nutzen wir verschieden methoden. Dies werden nachfolgend aufgeführt und erklärt.

Module und Information hiding

video

Um praktische Anwendungen zu programmieren müssen wir nicht immer die ganze Komplexität darstellen. Für andere Mensche die unseren Code verwenden, nicht es nicht hilfreich die genau funktionsweise aller methoden zu kennen, weshalb wir Code so strukturieren das wir nicht alles direkt erkennen könne.
Als beispielt gilt die autoschaltung. Wir können mit den Zahlen bei der Schaltung umgehen aber wissen nicht genau was passiert wenn wir in de 2. Gang schalten. Und das müssen wir auch nicht.

Packages und Imports

Packages sind dazu da Klassen in Java zu ordnen. Dafür wird das Dateisystem genutzt. Zusammengehörende Klassen müssen im gleichen Verzeichnis stehen. Der Name der Packages muss dem des Verzeichnisses entsprechen und muss in Quellcode vorhanden sein.
Angenommen wir haben drei Klassen A1, A2 und B1. Wir möchten, dass die Klassen A1 und A2 Teil eines Packages mit dem Namen foo sind. Die Klassen B soll ein Teil eines Packages mit dem Namen bar sein. Wir hätten folgende Dateien und Verzeichnisse in unserem Projekt:

Listing 32: packages

```
./src/foo/A1.java
./src/foo/A2.java
./src/bar/B1.java
```

Der Quellcode sieht dann enstprechend aus.

Listing 33: quellcode

```
package foo; // Paketdefintion, am Anfang der Datei

class A1 {
```

```
  // Implementation der Klasse
}
```

Hierarchien sind daruch auch möglich. Es können auch packages in packages verschatelt sein. das sieht dann so aus.

Listing 34: verschatelt packages

```
./src/foo/A1.java
./src/foo/A2.java
./src/foo/sub1/A31.java
./src/foo/sub2/A41.java

package foo.sub1;

class A31 {
  // Implementation der Klasse
}
```

Packages als Namesräume

Durch die Anordnung in Packages haben wir den Klassen nun auch neue Namen gegeben. Die Klasse A1 im obigen Beispiel hat nun den vollen Namen foo.A1. Die Klasse A31 hat den Namen foo.sub1.A31. Damit wird es möglich, denselben Klassennamen in verschiedenen Packages zu nutzen. Durch den vollen Namen können diese voneinander unterschieden werden. In grossen Programmen, mit zehntausenden von Klassen, ist dies enorm wichtig. Die Programmiererinnen können sich auf das Paket konzentrieren, indem sie gerade arbeiten. Sie können einen möglichst sinnvollen Klassennamen im aktuellen Paket wählen. Es kommt zu keinen Problemen mit Namen von Klassen in anderen Packages.
Dafür werden die Klassenname grösser. Beim nutzen sieht das jetzt so aus:

Listing 35: klassen in packages

```
package bar;
class B1 {
    // Felder
    foo.sub1.A31 field1;
    foo.sub2.A32 field2;

    B1() {
```

```
        this.field1 = new foo.sub1.A31();
        this.field2 = = new foo.sub2.A32();
    }
}
```

Java vereinfacht die Klassennamen indem es:

- annimt, wenn kein package angegeben ist, die Klasse im aktuellen Package vorhanden ist.
- Mit der Importklausel könne wir über kürzere namen verfügen

Importieren von Klassen

Um Klassen direkt ansprechen zu könne ohne noch den Packagenamen zu nutzen, müssen wir das Schlüsselwort **import** verwenden.

Listing 36: import Klassen

```
package bar;

import foo.sub1.A31;
import foo.sub2.A32;

class B1 {
    // Felder
    A31 field1;
    A32 field2;

    B1() {
        this.field1 = new A31();
        this.field2 = = new A32();
    }
}
```

Um alle Klassen eines Package zu importieren schreiben wir * ans ende des Package.

Bibliotheken in Java

Bibliotheken in java sind nichts weiteres als Sammlungen von Klassen, die in Packages strukturiert sind. Zum beispiel die Klasse ArrayList. Sie zu importieren geht so:

Listing 37: ArrayList

```
import java.utils.ArrayList
```

Module

Wie bereits angemerkt bietet Java noch eine zusätzliche Funktionalität, um Programme zu organisieren. Diese werden Module genannt. Jedes Modul beinhaltet eine Sammlung von Paketen. Die oben gezeigten Pakete in der API-Dokumentation befinden sich alle im Module java.base. Es gibt jedoch noch eine Vielzahl anderer Module. Diese bestehen dann auch wieder aus dutzenden von Paketen mit hunderten von Klassen. Welche Module in der aktuellen Java Version zur Verfügung stehen, können Sie auch in der Java API-Dokumentation nachschauen.
Da Sie auf absehbare Zeit keine so grossen Programme schreiben werden und mit diesem Mechanismus auch nicht direkt in Berührung kommen werden, werden wir Module in diesem Kurs nicht diskutieren.

Zugriffsmodifikatoren

Für Klassen gibt es die Zugriffsmodifikatoren **public** und **default**. Um eine Klasse als public zu deklarieren schreiben wir einfach das Schlüsselwort public vor die Klasse:

Listing 38: public

```
public class AClass {}
```

Wenn keine angaben zum Zugriff gemacht werden. Wird von **default** ausgegange
Das sind die Sichtbarkeitsregelen für Klassen

Zugriffsmodifikator	Sichtbarkeit
public	Der Zugriff ist aus Klassen im Programm möglich.
default	Der Zugriff ist nur aus Klassen im selben Package möglich.

Wir deklarieren nur die wichtigen Klassen, die von aussen Sichtbar sein sollen als public. Alle anderen Klassen, die nur der internen Implementation der Logik dienen, sind so vor den Benutzern versteckt. So könne wir auch teile des Codes Ersetzten ohne mit anderem Code zu intervenieren.

Zugriffsmodifikatoren für Felder und Methoden

Für Felder und Methoden gibt es noch mehr Zugriffsmodifikatoren. Hier auch wenn nichts deklariert wird, geht java von **default** aus.

Zugriffsmodifikator	Sichtbarkeit
public	Zugriff aus allen methoden möglich
protected	Zugriff aus Methoden von selben Klassen, Package und aus Subklassen möglich.
default	Der Zugriff ist nur aus Methoden von Klassen aus demselben Package möglich.
private	Der Zugriff ist nur aus Methoden von derselben Klasse möglich

Ein typisches Muster in der Objektorientierung ist es, die Felder alle als private zu deklarieren. Der Zugriff von Aussen wird nur über Methoden gewährt. Die Idee dahinter ist, dass die Datenrepräsentation ein internes Detail ist und sich häufig ändern kann. Im Gegensatz dazu wird das Verhalten, welches über die Methoden definiert ist, stabiler bleiben und sich weniger häufig ändern.
genaue Funktionalitaet der Zugriffsmodifikatoren

Generics

```
interface List<E> {
    // adds an Element to the list
    void add(E element);

    // returns the element at the given position
    E get(int index);
}
```

E steht für einen fixen Typen der aber noch nicht bekannt ist.

Typparameter

```
// Deklariert Variable vom Type List<String>
List<String> stringList;
```

Definieren einer Liste mit Strings als inhalt.

```
List<Double> reverse(List<Double> l) {
    // Implementation
}
```

Implementation wenn wir genauen Typ zurück wollen.

```
// Deklariert Variable vom Typ List<Integer> und initialisiert diese mit einer leeren
List<Integer> intList = new LinkedList<Integer>();
```

Erzeugen einer Instanz mit Generics.
Wir dürfen keine primitiven Typen im Konzept des Generics verwenden. Heisst **List¡int¿** geht nicht. Dies macht aber nichts aus, weil java automatisch primitive Datentypen in Referenzdatentypen umwandelt. Deshalb geht der Nachfolgend code.

```
List<Integer> list = new LinkedList<Integer>();
int e = 5;
list.add(e); // e wird automatisch in einen Integer umgewandelt.
```

Mehrere Typparameter

Wir könne auch Mehrere generische Parameter verwenden.

```
class Tuple<T, S> {
    T first;
    S second;

    Tuple(T first, S second) {
        this.first = first;
        this.second = second;
    }
}

Tuple<String, Integer> stringIntegerTuple = new Tuple<String, Integer>("a string", 5
```

Typeinschränkungen

Um mehr über einen Typ annehemen zu könne, könne wir engeben, das ein Typ ein Subtyps eines anderen Typs sein muss.

Listing 39: Comparble

```
interface Comparable<T> {
    int compareTo(T other);
}
```

```
class Integer implements Comparable<Integer> {
    int value;

    public int compareTo(Integer other) {
        if (value == other.value) {
            return 0;
        } else if (value < other.value) {
            return -1;
        } else {
            return 1;
        }
    }
}
```

Listing 40: typeinschränkung angewendet

```
class OrderedTuple<E extends Comparable<E>> {

    E value1;
    E value2;

    OrderedTuple(E value1, E.value2) {
        if (value1.compareTo(value2) == -1)  {
            this.value1 = value1;
            this.value2 = value2;
        } else {
            this.value1 = value2;
            this.value2 = value1;
        }
    }
}
```

Um einen Typ einzuschränken, den wir vergleichen wollen, müssen wir das Interface **Comparble** implementiere. Wenn wir zwei typen haben und sicher sein wollen das diese vergelichbar sind. Muss der eine Typ, **Comparble** extenden und den anderen Typen dort angeben oder wenn es ein Typ ist der mit sich selbst verglichen werden soll muss dort der eigen Typ drin stehen.
Die Methode **compareTo** gibt -1, 0, +1 zurück, abhängig von welcher Typ grösser als der andere ist.

Generische Methoden

Wir können auch Methoden generisch machen. Das wir oft bei Klassenmethoden angewendet. Hier ein Beispiel:

Listing 41: return first

```
class Util {

    public static <E> E extractFirst(List<E> list) {
        if (list.size() > 0) {
          return list.get(0);
        } else {
          return null;
        }
    }
}
```

Die methode gibt das erste elemte aus einer generischen Liste zurück. Wir geben nach den Modifikatoren (**public, static**) an, das die Methode generisch ist mit **¡E¿**

Listing 42: anwendung

```
class Main {
    public static void main(String[] args) {
        List<Integer> l = new LinkedList<Integer>();
        Util.extractFirst(l); // Aufruf der generischen Methode
    }
}
```

Wir könne auch bei Methoden Typeinschränkungen machen:

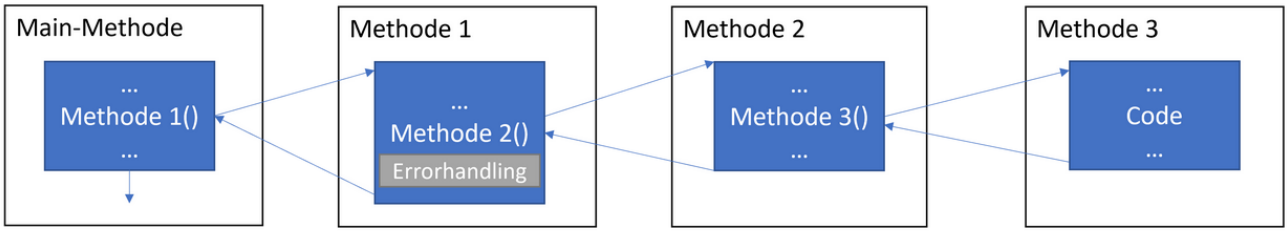
```
class Util {
    public static <T extends Comparable<T>> T max(T e1, T e2) {
      if (e1.compareTo(e2) < 0) {
          return e2;
      } else {
          return e1;
      }
    }
}
```

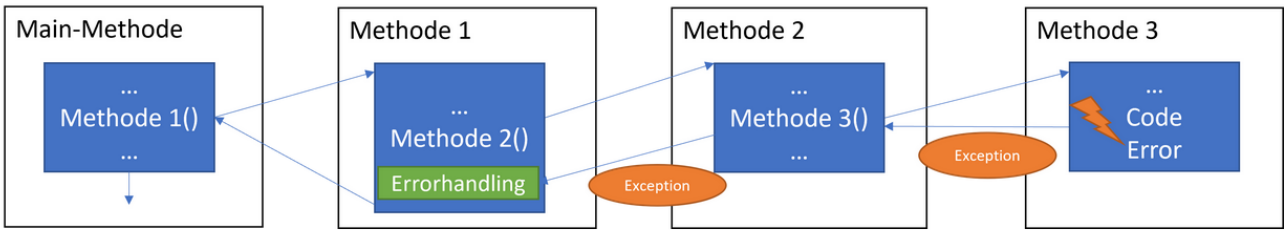
Fehlerbehandung und IO

Es gibt zwei Methoden Fehler zu behandeln. Einerseits mit **Exceptions**, die uns die möglichkeit geben mir dingen um zu gehen die das Programm nicht erwartet. Z.b eine Datei die gelöscht worden ist.
Es gibt aber auch Feheler die sehr oft vor kommen. z.b das wir in einer Liste eine Element suchen welches es nicht gibt. Solche Feheler werden von Java als spezeille Rückgabewert definiert, dieser heisst **Optional**

Exceptions

Exceptions sind Objekte die Information zum aufgetretenen Fehler enthalten. Wenn ein Fehler erkannt wird, wird dieser **gworfen**. Und Java gibt die Kontrolle der aufgerufenen Methode ab, falls diese diesen Feheler **fängt**, also eine Fehlerbehandung macht, geht dass programm wie erwartet weiter. Falls nicht wird die Exception zur nächsten Methode witere geben. Bis entweder sie gefangen wird oder in der main-Methoden ankommt und dört das Programm abstürtzt.





Exceptions sind einfach Klasse die von der Klasse Exceptions erben. Java stellt uns schon ein paar Exceptions zu verfügung aber, falls wir selber eine ausgabe definieren wollen könne wird das z.b so mache:

```
class MyException extends Exception {
    MyException(String errorMessage) {
        super(errorMessage);
    }
}
```

Im Konstruktor müssen wir jeweils den Superklassenkonstruktor (also den Konstruktor der Klasse Exception) aufrufen. Dieser nimmt als Argument einen String entgegen, der die Fehlermeldung enthält.
Wenn eine Fehelr im Code auftritt erstellen wir eine Instanz einer Exceptions und werfen die mit **throw**

```
if (unexpectedSituation) {
    MyException myException = new MyException("something unexpected happens");
    throw myException;
}
```

Wenn eine Methode eine Exception werfen kann, müssen wir das angeben. Die geschieht auch mit dem Wort **throw**. Dannach führen wir einfach auf, welche Exceptions geworfen werden können.

```
void exampleMethod() throws MyException {
    // ...
    if (unexpectedSituation) {
        MyException myException = new MyException("something unexpected happens");
        throw myException;
    }
    // ...
}
```

Wenn eine Methoden eine wetere Methode aufruft die einen Fehler werfen kann. Müssen wir diesen entweder behandeln oder an den aufrufenden weitergeben.
Wenn wir den Fehler behandeln wollen müssen wir ihn Fangen:

Try catch

```
try { // Start des geschuetzten Bereichts
    // ...
    exampleMethod() // hier kuennte ein Fehler auftreten
    // ...
} catch (Exception e) {
    // Fehlerbehandlung
}
```

Die Methode, welche einen Fehelr wefen kann, wird mit einem **try**-block umfasst und im **catch**-block wird der Feheler behandelt indem auf die Variable e zugegriffen werden kann.

finally Klausel

Die finally Klausel wird immer ausgeführt und ist dazu da um dinge aus zu führen die in einem Fehlerfall wie auch sonst passieren müssen. z.b wenn dateien erzeug wurde die wieder gelöscht werden müssen. Java garantier sogar das asus führen der finally-klausel auch wenn im catch-block wieder ein Fehler aufgetreten ist.

```
try { // Start des geschuetzten Bereichts
    // ...
    exampleMethod() // hier koennte ein Fehler auftreten
    // ...
} catch (Exception e) {
    // Fehlerbehandlung
} finally {
    // Code der immer ausgefuehrt werden muss
}
```

RuntimeExceptions könne immer auftreten weshalb sie nicht und die von ihnen abgeführten Fehler nicht deklariert werden müssen.

Optionals

Es gibt Situationen in welchen eine Fehler zu erwarten ist.

```
class Employees {
    public Employee getEmployeeWithId(int id) { /* ...*/  }
}
```

Falls es keinen angestellten mit der ID gibt, gibt es einen Fehler. Dies ist zu erwarten und wir könnten einen Fehler werfen. Da dies aber zu erwarten ist, sollte das mit der "normalen" Pogrammierlogik gelöst werden.
Wir könnten null zurück geben, fall der angestellten nicht existiert. Das hat aber viele Probleme. Einerseits erkennen wir das an der methode nicht das ein Feheler auftreten kann und mit **null** könne wir nicht viel anfangen.
Die Klasse **Optional** gibt uns eine bessere Lösung. Die ganze Klasse Eine grobe Definition der klasse:

```
class Optional<T> {
    T value = null;

    private Optional(T value) {
```

```
        this.value = value;
    }
    static <T> Optional<T> of(T value) {
        return new Optional<T>(value);
    }

    static <T> Optional<T> empty() {
        return new Optional<T>(null);
    }

    boolean isPresent() {
        return this.value != null;
    }

    public T get() {
        if (value == null) {
            throw new NoSuchElementException("No value present");
        }
        return value;
    }
}
```

Wir sehen, dass ein Optional entweder durch die Methode Optional.empty() oder Optional.of() kreiert werden kann. Im ersten Fall wird ein Optional erzeugt, dass keinen Wert enthält, im zweiten Fall wird der übergebene Wert gespeichert. Mittels der isPresent Methode kann abgefragt werden, ob das Optional einen Wert enthält. Die get Methode wird benutzt, um den Wert zu erhalten

```
class Employees {
    public Optional<Employee> getEmployeeWithId(int id) { /* ...*/  }
}

int id = 5;
Optional<Employee> maybeAnEmployee = employees.getEmployeeWithId(id);
if (maybeAnEmployee.isPresent()) {
    Employee employee = maybeAnEmployee.get()
   // Normale Programmlogik
} else {
   // Logik fuer den Fall, dass Employee nicht gefunden wurde.
```

```
}
```

Diese Lösung hat grosse Vorteile. Erstens sehen wir an der Methodensignatur Optional¡Employee¿ getEmployeeWithId(int id), dass diese Methode eventuell keinen Wert zurückliefert. Zudem können wir die Fehlerbehandlung gar nicht versehentlich vergessen, da wir ja nicht direkt auf den Wert employee zugreifen können, ohne zuerst get() aufzurufen. Der Java-Compiler wird uns anderenfalls eine Fehlermeldung geben, und uns daran erinnern, dass wir noch eine Fehlerbehandlung machen sollten.

Funktionales Programmieren

Funktionales Programmieren ist eine andere Art der Programmierlogik, im vergleich zum Objektorientierten Programmieren. Beim Funktionale Programmieren werden funktionen im mathematischen sinne betrachtet. $f : I \to O$ Wenn wir eine eingabe machen krigen wir einen neuen Wert herraus.
Nehmen wir an wir haben eine funktion $f : I \to R$ und eine funktion $g : R \to O$. So könne wir ein grösseres programm $h(x)$ schreiben.

$$h : I \to O$$

$$h(x) = g(f(x))$$

Ein konretes Beispiel:

```
class FunctionalTest {
    static double square(double x) {
        return x * x;
    }

    static int round(double number) {
        return (int) (number + 0.5);
    }

    static int roundSquared(double x)  {
        return round(square(x));
    }
}
```

Diese Hintereinanderausführung von Funktionen klingt zuerst einmal nicht besonders spannend. Ein weiteres Element, welches die funktionale Programmierung sehr mächtig macht, ist, dass

darin Funktionen einfach Werte sind. Damit können diese zum Beispiel als Parameter an andere Funktionen (Methoden) übergeben werden. Oder sie können als Rückgabewert aus einer Methode zurückgegeben werden. Wie wir noch sehen werden, macht es dies möglich, gewisse Arten von Berechnungen sehr elegant und verständlich auszudrücken.
Ein weiteres Merkmal von diesem Programmierstil ist, dass Daten von einer Funktion nicht verändert werden. In der objektorientierten Programmierung wird oft das Verändern von Daten nur von Methoden eines Objektes erlaubt. Im Gegensatz dazu wird es in der funktionalen Programmierung gänzlich vermieden. Stattdessen werden die Eingabedaten einer Funktion wenn nötig kopiert, und nur diese Kopie wird verändert. Dies macht das Verstehen von Programmen einfacher, da wir nicht im Kopf Buch führen müssen, welche Daten von welcher Funktion verändert wurden.

Funktionsobjekte

Eine Zentrale idee des funktionalen Programmierens ist, das Funktionen Werte sind. Funktionen könne also als Variabeln gespeichert werden und so auch als Parameter anderen Funktione übergeben und auch aus Funktionen zurück gegeben werden. In Java simulieren wir dies indem wir Funktionen als Objekte repräsentieren.

Listing 43: funktion

```
interface Function<T, R> {
    R apply(T x);
}
```

Dieses Interface bildet eine Funktion $f : R \to T$. T ist der eingabewert und R der Rückgabewert. So könnte wir die methode **square** implementieren.

Listing 44: square als Funktion

```
class SquareFunction implements Function<Integer, Integer> {
    public Integer apply(Integer x) {
        return x * x;
    }
}
```

Objekte dieser Funktionen können nun an Methoden übergeben werden:

```
class FunTest {
    static int valueAtX (Function<Integer, Integer> fun, Integer x) {
        return fun.apply(x);
```

```
    }

    public static void main(String[] args) {
        int squareOf2 = valueAtX(new SquareFunction(), 2);
        int cubeOf2 = valueAtX(new CubicFunction(), 2);
    }
}
```

Listing 45: square und cube als funktion

```
    interface Function<T, R> {
    R apply(T x);
}

class PowerFunction implements Function<Integer, Integer> {
    int n = 0;

    public PowerFunction(int n) {
        this.n = n;
    }

    public Integer apply(Integer x) {
        int pow = 1;
        for (int i = 0; i < n; i = i + 1) {
            pow = pow * x;
        }
        return pow;
    }
}

class Main {
   static int valueAtX (Function<Integer, Integer> fun, Integer x) {
        return fun.apply(x);
    }

    static Function<Integer, Integer> nthPower(int n) {
      return new PowerFunction(n);

    }
```

```
  public static void main(String[] args) {
    Function<Integer, Integer> squareFunction = nthPower(2);
    Function<Integer, Integer> cubicFunction = nthPower(3);

    System.out.println("square at 2: " + valueAtX(squareFunction, 2));
    System.out.println("cube at 2: " + valueAtX(cubicFunction, 2));
  }
}}
```

Lambda-Ausdrücke und Methodenreferenzen

Im **java.util.function** package ist dieses Interface enthalten, welche eine Funktion simuliert:

```
interface Function<T, R> {
    R apply(T x);
}
```

So würden wir mit dem Interface **Function** eine Funktion definieren:

```
class SquareFunction implements Function<Integer, Integer> {
    public Integer apply(Integer value) {
        return x * x;
    }
}

Function<Integer, Integer> squareFun = new SquareFunction();
```

Das alles könne wir mir Lambda-Ausdrücken auf ein Zeile schreiben.

```
Function<Integer, Integer> squareFun = (Integer x) -> { return x * x };
```

Allgemeiner fomuliert entspricht der lambda-asudruck **(T value) -¿ Ausdruck**:

```
class LambdaExpression implements Function<Integer, Integer> {
    public R apply(T value) {
        AUSDRUCK;
    }
}
```

Vieles könne wir an diesem Ausdruck noch verreinfachen. Wir können die Datentypen, die Klammern und das **return** weglassen. Dann sieht das so aus:

```
Function<Integer, Integer> squareFun = x -> x * x;
```

Es ist auch möglich zwei argumente für eine Funktion zu verwenden. Das Interface dafür heisst **BiFunction**:

```
interface BiFunction<T, U, R> {
    R apply(T x, U y);
}
```

Und wird so ausgeführt

```
BiFunction<Integer, Integer, Integer> plusFun = (Integer a, Integer b) -> a + b;
```

Methodenreferenzen

Neben Lambda-Ausdrücken können auch Methoden mit der richtigen Signatur als Wert behandelt werden. Eine Methode mit nur einem Argument könnte also einer Variable mit Typ Function¡T, R¿ zugewiesen werden. Analog dazu können wir eine Methode mit zwei Argumenten einer Variable mit Typ BiFunction¡T, U, R¿ zuweisen. Um die Methode als Wert zu behandeln, schreiben wir den Objektnamen, gefolgt von :: und dem Methodennamen. Dies ist im Folgenden illustriert:

```
class MethodExample {
    Integer square(Integer x) {
        return x * x;
    }

    public static void main(String[] args) {
        MethodeExample instance = new MethodeExample();

        Function<Integer, Integer> squareFun = instance::square;
    }
}
```

Und das ganze noch mit Klassen

```
class MethodExample {
    static Integer square(Integer x) {
        return x * x;
    }

    public static void main(String[] args) {

        Function<Integer, Integer> squareFun = MethodeExample::square;
    }
```

```
}
```

Und insgesamt:

```
import java.util.function.Function;


class MethodExample {
    Integer square(Integer x) {
        return x * x;
    }

    static Integer squareStatic(Integer x) {
        return x * x;
    }

}

public class Main {
    public static void main(String[] args) {
        MethodExample instance = new MethodExample();

        Function<Integer, Integer> squareFun = instance::square;

        System.out.println(squareFun.apply(7)); //gibt 49


        Function<Integer, Integer> squareFun2 = MethodExample::squareStatic;

        System.out.println(squareFun2.apply(7)); // gibt 49


    }
}
```