
UNIVERSITÄT
LEIPZIG

# EOOMP

Übung 7

# Fragen?

1. **Eigene Exceptions erstellen**
2. **Generics & Collections**

# EIGENE EXCEPTIONS IMPLEMENTIEREN

# GENERICS & COLLECTIONS

# WDH: ARRAYS

Wir haben bereits viel mit Arrays gearbeitet und gesehen, dass man sie **einfach** anlegen kann und **schnell** auf Elemente zugreifen kann bzw. diese ändern kann.

**Aber** wir haben auch gesehen:

- Arrays haben eine **feste Größe**! Aufgaben wie „Füge ein zusätzliches Element ein“ gingen nur umständlich.
- Beim Deklarieren legt man einen Datentyp fest, welcher **unveränderlich** ist.
- Alle Elemente im Array müssen den **gleichen Datentypen** haben.
  (Es dürfen auch Instanzen von Unterklassen sein)

*Was ist denn nun, wenn ich ein Array von Weihnachtsbaumkugeln anlege*

*und später im Keller eine weitere Box mit Kugeln finde??*

# COLLECTIONS

Dafür gibt es in Java die sogenannten **Collections**. Diese sind **flexible Datenstrukturen**, die ebenfalls dazu dienen **mehrere Objekte zu speichern** und *effizient* zu **bearbeiten**.

**Vorteile von Collections**

- Sie können **automatisch wachsen**, d.h. man muss die Größe vorher nicht angeben.
- Es gibt fertige **Hilfsmethoden**, z.B. suchen, sortieren, filtern, kombinieren, ...

In Java gibt **drei Haupttypen** von Collections: `List`, `Set`, `Map`.

# COLLECTIONS

| List | Set | Map |
| --- | --- | --- |
| • **Geoordnete** Sammlung von Elementen, die **Duplikate erlaubt**.<br>• Der Zugriff auf Elemente erfolgt über **Indizes**.<br>(Aber nicht mehr mit [i], sondern mit set() und get()! ) | • Eine Sammlung von eindeutigen Elementen.<br>• Sie **kann geordnet** sein (z.B. TreeSet) oder ungeordnet sein. (z.B. HashSet)<br>• Es sind **keine Duplikate** erlaubt.<br>(vergleichbar mit mathematischen Mengen) | • Eine Sammlung von **Schlüssel – Wert-Paaren.**<br>• Bei den **Schlüsseln** sind **keine Duplikate** erlaubt<br>• Bei den **Werten** sind **Duplikate erlaubt.**<br>• Der Zugriff auf Elemente erfolgt über die Schlüssel. |
| ArrayList | HashSet, TreeSet | HashMap, TreeMap |

# COLLECTIONS – ETWAS TECHNISCHER

In Java ist `java.util.Collection` ein zentrales **Interface** für alle standardmäßigen Datenstrukturen, die eine Gruppe von Objekten repräsentieren. Es enthält die grundlegenden Operationen wie `add()`, `remove()`, `contains()`, `size()`, etc.

→ Aber Achtung, als Interface sind hier nur die Methoden-Deklarationen angegeben!

`List` und `Set` sind **Unter-Interfaces** von `Collection`! Das bedeutet sie erben alle Methoden(-Deklarationen) von `Collection` und spezifizieren zusätzliche Eigenschaften.

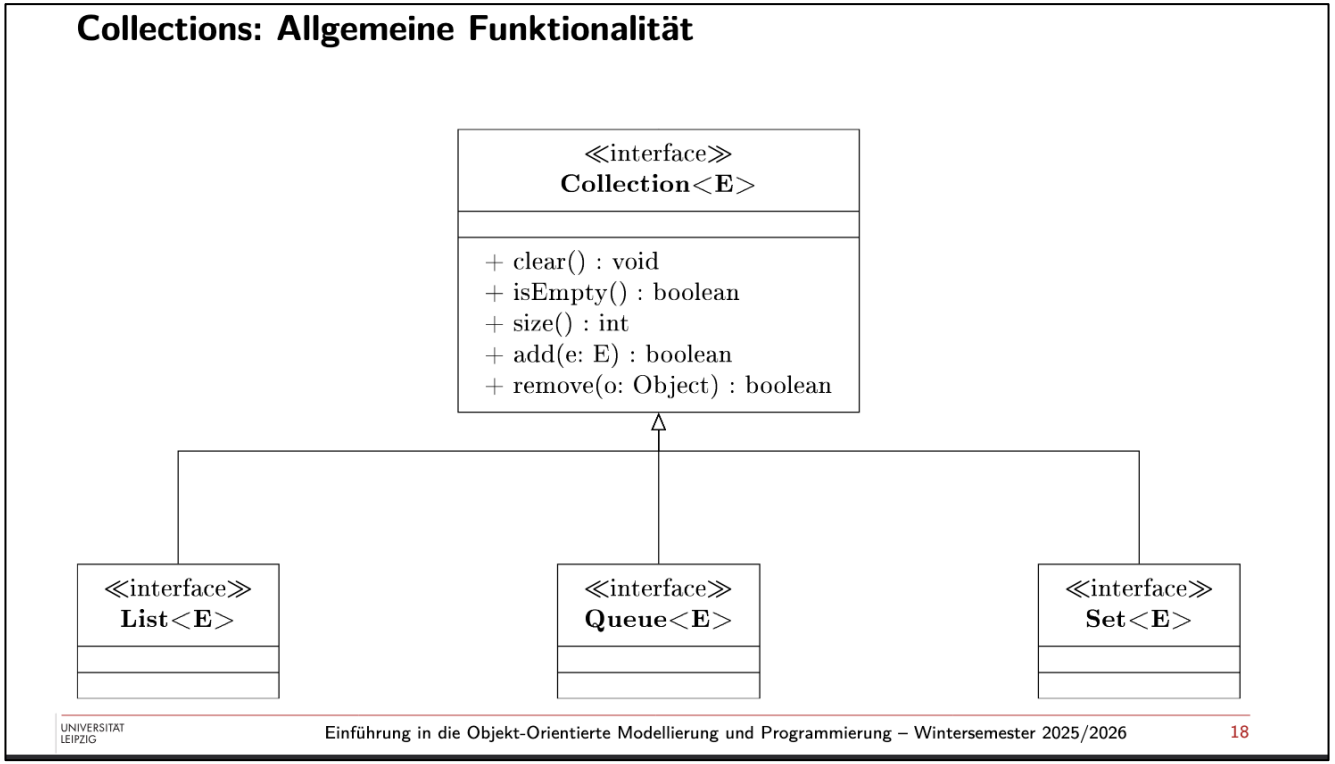
→ Immer noch keine Implementierung der Methoden!

Eine typische **Implementierung** von `List` ist `ArrayList`.

→ Hier werden also diese Methoden erst implementiert!

`Map` zählt zwar zu den Collections in Java, ist aber **kein Unter-Interface** von `Collection`.

# COLLECTIONS – ETWAS TECHNISCHER



# GENERICS

Collections benötigen nun aber ebenfalls einen Datentyp.

→ **Generics** sind sogenannte **Platzhalter für Datentypen**.

→ Collections sind dann **invariant**, dadurch haben sie eine höhere Typsicherheit (Datentypfehler werden bereits zur Compile-Zeit erkannt)

→ Man kann Code nun „noch allgemeiner“ schreiben, um ihn später wieder zu verwenden.
Mit Generics kann man z.B. eine Liste einmal implementieren und später sowohl für Integer, String oder eigene Klassen verwenden.

# GENERICS – EINFACHES BEISPIEL

MeineGenerischeKlasse.java

```java
public class MeineGenerischeKlasse<T> {

    //Attribute
    private T wert;

    //Methoden
    public void set(T w)
    {
        this.wert = w;
    }

    public T get()
    {
        return this.wert;
    }
}
```

Main.java

```java
public class Main {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {

        MeineGenerischeKlasse<Integer> typ1 = new MeineGenerischeKlasse<>();

        typ1.set(w: 3);
        int i = typ1.get();

        System.out.println(i);

        //-----------------------------------------------------------------

        MeineGenerischeKlasse<String> typ2 = new MeineGenerischeKlasse<>();

        typ2.set(w: "Hallo!");
        String s = typ2.get();

        System.out.println(s);
    }
}
```

Ausgabe

```
3
Hallo!
```

Achtung: bei primitiven Datentypen hier die „Wrapper Klasse“ einsetzen!

# ARRAYLIST

```
ArrayList<Double> mylist = new ArrayList<>();
```

- `mylist.size()` → bestimmt die Länge der Liste (bei Arrays war es `meinarray.length`)
- `mylist.add(3.2)` → fügt das Element 3.2 am Ende der Liste hinzu
- `mylist.add(2, 6.5)` → fügt das Element 6.5 an der 3. Stelle der Liste ein (beginnt bei 0 zu zählen)
- `mylist.contains(2.1)` → gibt wahr/falsch zurück, ob das Element 2.1 in der Liste vorkommt

# ARRAYLIST

```
ArrayList<Double> mylist = new ArrayList<>();
```

- `mylist.get(3)` → gibt das 4. Element der Liste zurück
- `mylist.set(8, 1.2)` → ändert das 9. Element der Liste zu 1.2
- `mylist.remove(4)` → löscht/entfernt das 5. Element der Liste
- `mylist.remove(3.2)` → entfernt das **erste** vorkommende Element mit dem Wert 3.2)

# ARRAYLIST

```
ArrayList<Integer> mylist = new ArrayList<>();
```

Achtung! Wenn die `ArrayList` **Integer** speichern soll, dann kann bei Methoden wie **`remove`**() nicht unterschieden werden, ob der eingesetzte Wert der Index oder der Wert der Liste an einer Stelle sein soll, da beides Integer Werte sind!

**Trick:**

```
Mylist.remove(1)
Mylist.remove(Integer.valueOf(1))
```

→ entfernt 2. Element
→ entfernt Element mit Wert 1

# BEISPIELE

# AUFGABE 1

1. Erstellen Sie eine `ArrayList` namens `meineStringListe`, welche `Strings` speichern kann.
2. Fügen Sie die Elemente „Hugo“, „Heinrich“, „Helga“ hinzu.
3. Geben Sie alle Elemente von `meineStringListe` aus.
4. Löschen Sie den 2. Eintrag („Heinrich“)
5. Ändern Sie den 1. Eintrag zu „Herbert“.
6. Geben Sie erneut alle Elemente von `meineStringListe` aus.

# AUFGABE 2

1. Erstellen Sie eine `ArrayList` namens `meineIntegerListe`, welche Integer Werte speichern kann.
2. Fügen Sie alle Zahlen von 0 bis 100 hinzu.
3. Addieren Sie alle Elemente von `meineIntegerListe` und geben Sie das Ergebnis aus.
4. Fügen Sie den Wert 101 am Ende von `meineIntegerListe` hinzu.
5. Fügen Sie den Wert 333 an der 88. Stelle hinzu.
6. Löschen Sie den 65. Eintrag.
7. *Löschen Sie den Eintrag mit dem Wert 38.*
8. *Ändern Sie den 2. – 10. Eintrag, indem Sie zu dem vorhandenen Element 10 addieren.*
9. Addieren Sie wieder alle Elemente von `meineIntegerListe` und geben Sie das Ergebnis aus.

*(etwas schwieriger)*

Fragen bitte immer an

barz@informatik.uni-leipzig.de