Collections

Wir möchten uns heute mit Collections beschäftigen. Collections sind Objekte, die andere Objekte auf Grund einiger Regeln halten können.

Arten von Collections:

ArrayList:
Eine ArrayList ist eine Sammlung, die sich wie ein Array verhält aber keine Platzbegrenzung hat.

Map:
Eine Map ist eine Sammlung von zusammengehörigen Paaren.

Set:
Ein Set ist eine Sammlung in der jedes Element nur einmal vorkommen darf.

Iterator
Diese Klasse kann für Iterationen über die Sammlungen genutzt werden.

Syntax:

Die Erstellung solcher Sammlungen ist dazu da, Objekte zu verwalten. Java ist so gemacht, dass es immer prüft ob bestimmte Objekte immer regelkonform verwendet werden. Deshalb wurden die generischen Typen entwickelt. Das bedeutet bei jeder Sammlung muss nun der Typ des Elements welches gespeichert werden soll angegeben sein. Objekte anderen Typs können dadurch nicht mehr für die Sammlung verwendet werden. Das wird einfach durch die Anhängung von **< Objekttyp >** an die jeweilige Sammlung erreicht.

Zu beachten ist jedoch, dass Sammlungen an sich keine primitiven Datentypen speichern können. Dank Javas autoboxing und autounboxing ist es jedoch Möglich. Deshalb können wir auch primitive Datentypen wie ints und doubles in Sammlungen stopfen, unter der Vorraussetzung die Klasse des primitiven Typen in die Spitzklammern zu schreiben.

int = Integer
double = Double
boolean = Boolean
char = Character
float = Float
long = Long
short = Short

ArrayList

Die ArrayList verhält sich genauso wie ein Array, nur dass sie keine Platzbegrenzung hat. Ein weiterer kleiner Unterschied ist die Syntax und dass man bei der ArrayList mit ihren Methoden arbeiten muss.

Deklaration

```
ArrayList< Objekt > listName = new ArrayList< Objekt >();
```

Die ArrayList kann Objekte aller Klassen halten.

Zu beachten ist jedoch, dass Sammlungen an sich keine primitiven Datentypen speichern können. Dank Javas autoboxing und autounboxing ist es jedoch Möglich. Deshalb können wir auch primitive Datentypen wie ints und doubles in Sammlungen stopfen, unter der Vorraussetzung die Klasse des primitiven Typen in die < > zu schreiben.

int = Integer
double = Double
boolean = Boolean
char = Character
float = Float
long = Long
short = Short

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

Hier wird eine ArrayList erstellt die beliebig viele ints speichern kann.

Nicht vergessen: Wenn man die ArrayList verwenden möchte dann muss man sie natürlich auch importieren:

```
import java.util.ArrayList;
```

Wichtige Methoden zur Benutzung einer Arraylist:

Das Objekt der ArrayList muss natürlich von derselben Klasse sein wie das Objekt das in den Parametern der Methoden verwendet wird.

boolean add(Object obj)

Fügt das Objekt an dem Ende der Liste hinzu. Diese Methode gibt true zurück, wenn das Element erfolgreich der Liste angehängt wurde.

void add(int index, Object obj)

Fügt das Objekt bei der IndexNummer *index* in die ArrayList ein. Zu Beachten ist, dass Element, welches sich vorher an der Stelle befunden hat, und alle Elemente die sich hinter dem Element befunden haben1 werden eine Stelle nach hinten geschoben (+1 auf den Index).

Object remove(int index)

Löscht das Objekt an der Stelle *index*. Außerdem wird das gelöschte Objekt zurückgegeben um gegebenenfalls noch verwendet zu werden.

Object set(int index, Object obj)

Ersetzt das Object an der Stelle *index* mit dem Objekt *obj*, welches im Parameter übergeben wurde. Des Weiteren wird das alte Objekt zurückgegeben.

Object get(int index)

Gibt das Objekt an der Stelle *index* zurück.

int size()

gibt die Anzahl der Einträge die in der ArrayList gespeichert sind als int zurück.

boolean contains(Object obj)

Gibt true zurück wenn die Liste mindestens ein Objekt in der Liste hat, welches gleich mit dem Objekt im Parameter ist.

int indexOf(Object obj)

Gibt die Indexnummer des **ersten** Objekts, welches gleich dem Objekt im Parameter ist. Sollte kein solch Element vorhanden sein wird -1 zurück-gegeben.

int lastIndexOf(Object obj)

Gibt die Indexnummer des **letzten** Objekts, welches gleich dem Objekt im Parameter ist. Sollte das Element nicht vorhanden sein wird -1 zurück-gegeben.

Da die Methoden der ArrayList sehr einfach und selbsterklärend sind, gibt es hier nur einfach einen Auszug aus meiner Klasse Arraylist.

```
import java.util.ArrayList;

public class Arraylist {

    private ArrayList<String> namensListe;

    public Arraylist()
    {
        namensListe = new ArrayList<String>();
    }
    public boolean hinzufügen(String name)
    {
        return namensListe.add(name);
    }
    public void hinzufügenAnStelle(int index, String name)
    {
        namensListe.add(index, name);
    }
    public void löschen(int index)
    {
        namensListe.remove(index);
    }
    public void ersetzen(int index, String name)
    {
        namensListe.set(index, name);
    }
    public String gibName(int index)
    {
        return namensListe.get(index);
    }
    public int gibLängeDerListe()
    {
        return namensListe.size();
    }
    public boolean listeHatNamen()
    {
        return !namensListe.isEmpty();
    }
    public boolean hatNamen(String name)
    {
        return namensListe.contains(name);
    }
    public int nameAnStelle(String name)
    {
        return namensListe.indexOf(name);
    }
}
```

Iteration

Die Iteration über eine ArrayList kann mit einer einfachen foreach-Schleife vollbracht werden.

```
for (Object listObjects : arrayListName)
        machWasMit(listObjects);
```

Hier als Beispiel, eine Methode die alle eingetragenen Namen in der Namensliste in der Konsole ausgibt:

```
public void allesAusgeben()
{
        for (String namen : namensListe)
                System.out.print(namen + ", ");
}
```

Bei der obigen Methode würde der letzte Eintrag auch noch einen Beistrich angehängt bekommen. Das wollen wir ändern und mithilfe einer for-Schleife und der .size() Methode gehen wir vor.

```
public void allesAusgebenBesser()
{
        for (int index = 0; index < (namensListe.size()-1); index++)
                System.out.print(namensListe.get(index) + ", ");
        System.out.print(namensListe.get(namensListe.size()-1));
}
```

Die for-Schleife spricht alle bis auf das letzte Element an und gibt sie in der Konsole mit einem angehängten Beistrich aus. Dann wird noch das letzte Element (länge -1) ohne Beistrich ausgegeben.

Map / HashMap

Die Map an sich ist eine abstrakte Klasse, von der man keine Instanzen bilden kann, die man dann in seiner Klasse nutzen könnte. Um eine Map jedoch ordnungsgemäß verwenden zu können, wurde die HashMap erschaffen, von der man problemlos ein Objekt erstellen, und sie somit benutzen kann.

Deklaration:

```
HashMap< Key, Value > mapName = new HashMap< Key, Value >();
```

Das Key-Element und das Value-Element können Objekte jeder Klasse sein. Zu beachten ist jedoch, dass Sammlungen an sich keine primitiven Datentypen speichern können. Dank Javas autoboxing und autounboxing ist es jedoch Möglich. Deshalb können wir auch primitive Datentypen wie ints und doubles in Sammlungen stopfen, unter der Vorraussetzung die Klasse des primitiven Typen in die < > zu schreiben.

int = Integer
double = Double
boolean = Boolean
char = Character
float = Float
long = Long
short = Short

```
HashMap<String, Double> preisListe = new HashMap<String, Double>();
```

Hier wird eine HashMap erstellt die einen String als Key benutzt, dem ein double als Value zugewiesen wird. Wichtig, es wird die Klasse Double in die Spitzklammern geschrieben und nicht double, wie der primitive Datentyp heißen würde. Da die Entwickler von Java ihren so genannten Wrapper-Klassen fast den gleichen Namen gegeben haben, fällt es nicht auf. Diese Map z.B. kann als eine Preisliste fungieren, die zu jedem Artikelnamen (String) den Preis (double) speichert.

Wichtig: Die Import-Anweisung darf natürlich nicht vergessen werden. Um nicht die vielen anderen Sammlungen die höchstwahrscheinlich bei der Iteration zum Einsatz kommen werden zu übersehen empfiehlt sich:

```
import java.util.* ;
```

Wichtige Methoden zur Benutzung einer HashMap:

HashMap<Object, Object> map = new HashMap<Object, Object>();

Die Key und Value Objects müssen natürlich von derselben Klasse sein wie die Objects die in den Parametern der Methoden verwendet werden.

boolean containsKey(Object key)
gibt true zurück, wenn die Map einen Key gespeichert hat der derselbe ist wie der Parameter.

boolean containsValue(Object value)
gibt true zurück, wenn die Map eine Value gespeichert hat die dieselbe ist wie der Parameter.

Value get(Object key)
gibt das Value-Objekt zurück, dass zu dem Key gespeichert wurde. null falls der Key nicht existiert.

boolean isEmpty()
gibt true zurück wenn die Map leer ist.

Value put(Object key, Object value)
speichert den Key mit der dazugehörigen Value. Sollte der Schlüssel schon vorhanden sein, wird die alte Value einfach überschrieben. Gibt die alte Value zurück, oder null wenn noch kein Eintrag mit dem Key stattgefunden hat.

Value remove(Object key)
löscht den im Parameter angegeben Key und dadurch auch die dazugehörige Value. Gibt die Value des gelöschten Keys zurück, oder null wenn der Schlüssel nicht vorhanden ist.

int size()
gibt die Anzahl der Paare zurück.

void clear()
Löscht alle Einträge der Map.

Verwendung der Methoden anhand der Map *preisListe*:

```
HashMap<String, Double> preisListe = new HashMap<String, Double>();
```

Wenn der Artikel noch nicht vorhanden ist, wird er hinzugefügt.

```
public void artikelHinzufügen(String produkt, double preis)
{
	if(!preisListe.containsKey(produkt))
		preisListe.put(produkt, preis);
	else
		System.out.println("Dieser Eintrag ist schon vorhanden!");
}
```

Löscht ein Schlüsselpaar(Artikel) sofern es vorhanden ist.

```
public void artikelLöschen(String produkt)
{
	if (preisListe.containsKey(produkt))
		preisListe.remove(produkt);
	else
		System.out.println("Dieser Eintrag ist nicht vorhanden");
}
```

Ändert die Value(Preis) eines Eintrags sofern der Schlüssel schon existiert.

```
public void artikelÄndern(String produkt, double preis)
{
	if(preisListe.containsKey(produkt))
		preisListe.put(produkt, preis)
	else
		System.out.println("Dieser Eintrag ist nicht vorhanden");
}
```

Gibt die Größe (Anzahl der Schlüsselpaare) zurück.

```
public int wieVieleArtikel()
{
	return preisListe.size();
}
```

Gibt die Value(Preis) zu dem im Parameter angegebenen Key(Artikel) zurück.

```
public double artikelPreis(String produkt)
{
	return preisListe.get(produkt);
}
```

Iteration

Die Iteration beinhaltet 3 Formen:

HashMap<Object, Object> mapName = new HashMap<Object, Object>();

1) Iteration über die Schlüsselwerte der Map:
Mithilfe der Methode mapName.keySet(), die ein Set aller Keys bereitstellt.

```
Set<Object> keys = mapName.keySet();
        for (Object keyObjects : keys)
                machWasMit(keyObjects);
```

2) Iteration über die Valuewerte der Map:
Mithilfe der Methode mapName.values(), die eine Collection mit allen Valuewerten zurückgibt.

```
Collection<Object> values = mapName.values();
        for (Object valueObjects : values)
                machWasMit(valueObjects);
```

3) Iteration über Schlüssel- und Valuewerte der Map:
Die Methode mapName.entrySet() liefert ein Set mit einer EntryMap in der Schlüssel- und Valuewerte gespeichert sind, auf die dann mit .getKey() und .getValue() zugegriffen werden kann.

```
Set<Entry<Object, Object >> entries = mapName.entrySet();
        for (Entry< Object, Object > einträge : entries) {
                machWasMit(einträge.getKey());
                machWasMit(einträge.getValue());
        }
```

Die Objekte müssen natürlich immer mit den Objekten, die bei der Erstellung der HashMap verwendet worden sind, übereinstimmen.

HashMap<String, Double> preisListe = new HashMap<String, Double>();

Eine Methode die alle Artikel in der Konsole ausgibt:

```
public void artikelAusgeben()
{
Set<String> keys = preisListe.keySet();
        for (String artikel : keys)
                System.out.println(artikel);
}
```

Eine Methode die alle Preise summiert und davon den Durchschnitt bildet:

```
public double preisDurchschnitt()
{
double summe = 0.0;
Collection<Double> values = preisListe.values();
        for (Double preise : values)
                summe += preise;
return summe / preisListe.size();
}
```

Eine Methode die die Preisliste gesamt, und den Durchschnittspreis ausgibt:

```
public void allesAusgeben()
{
Set<Entry<String,Double>> artikelListe = preisListe.entrySet();
         for (Entry<String, Double> einträge : artikelListe)
                 System.out.println(einträge.getKey() + " : " + einträge.getValue());
        System.out.print("Preisdurchschnitt: "+ preisDurchschnitt());
}
```

Hier wird die ganze Map durchiteriert und alle Paare getrennt durch einen Doppelpunkt ausgegeben. Am Schluss wird noch die oben erstellte Methode aufgerufen um den Durchschnittspreis extra auszugeben.

Wenn bei der zu erstellenden Methode Key und Value einer Map in irgendeiner Weise vonnöten sind, sei es zur Abfrage, um direkt verändert zu werden oder in einer anderen Weise, führt nur die 3te Art der Iteration zum Ziel.

Um die komplizierten Iterationen leichter zu verstehen folgt nun eine Methode die von unserer Preisliste den Artikelnamen zurückgeben soll, der der teuerste ist (sozusagen das Maximum unter den Valuewerten).

```
public String teuersterArtikel()
{
	String artikel = "";
	double maximum = 0;
	Set<Entry<String,Double>> artikelListe = preisListe.entrySet();
	for (Entry<String, Double> einträge : artikelListe) {
	   if (einträge.getValue() > maximum) {
	      maximum = einträge.getValue();
	      artikel = einträge.getKey();
	   }
	}
	return artikel;
}
```

Diese Methode erstellt zuerst die Entrys *artikelListe*, die mit den Einträgen der *preisListe* durch .entrySet() gefüllt wird. Danach wird mithilfe einer foreach Schleife darüber iteriert. Wenn ein Schlüsselpaar einen Valuewert hat, der größer als das derzeitige Maximum ist, wird das Maximum zu dem jetzt größeren Wert und der Schlüssel(der Artikel) wird in *artikel* zwischen gespeichert. Wenn die Schleife fertig ist, ist der teuerste Artikel in *artikel* gespeichert und wird zurückgegeben.

Alle Artikel zurückgeben die einen bestimmten Preis haben:

```
public ArrayList<String> artikelMitPreis(double preis)
{
ArrayList<String> artikel = new ArrayList<String>();
Set<Entry<String,Double>> artikelListe = preisListe.entrySet();
	for (Entry<String, Double> einträge : artikelListe)
		if (einträge.getValue() == preis)
			artikel.add(einträge.getKey());
return artikel;
}
```

Wenn der Preis(Value) eines Schlüsselpaares gleich dem Parameter ist, wird der Artikelname(Key) in die zu zurückgebende ArrayList gepackt.

Set / HashSet

Das Set an sich ist eine abstrakte Klasse, von der man keine Instanzen bilden kann, die man dann in seiner Klasse nutzen könnte. Um das Set jedoch ordnungsgemäß verwenden zu können, wurde das HashSet erfunden, von dem man problemlos ein Objekt erstellen, und es somit benutzen kann.

Deklaration:

```
HashSet< Object > setName = new HashSet< Object >();
```

Das HashSet kann Instanzen jeder Klasse speichern. Zu beachten ist jedoch, dass Sammlungen an sich keine primitiven Datentypen speichern können. Dank Javas autoboxing und autounboxing ist es jedoch Möglich. Deshalb können wir auch primitive Datentypen wie ints und doubles in Sammlungen stopfen, unter der Vorraussetzung die Klasse des primitiven Typen in die < > zu schreiben.

int = Integer
double = Double
boolean = Boolean
char = Character
float = Float
long = Long
short = Short

```
HashSet<Integer> zahlenMenge = new HashSet<Integer>();
```

Hier wird ein Set erstellt, welches Zahlen speichern soll. Es kann allerdings jede Zahl nur einmal speichern. Das bedeutet in diesem Set wird z.B. die Zahl 5 höchstens einmal vorkommen. Für Objekte gilt:

Sollte *objekt1.equals(objekt2)* true ergeben, wird nur eines der beiden im Hashset gespeichert, da in einer Menge jedes Element nur ein Mal vorkommen darf.

Um das HashSet verwenden zu können muss folgende import-Anweisung geschrieben werden:

```
import java.util.HashSet;
```

Wichtige Methoden zur Benutzung eines HashSets:

Das Objekt des HashSets muss natürlich von derselben Klasse sein wie das Objekt das in den Parametern der Methoden verwendet wird.

boolean add(Object obj)
Fügt das Objekt, welches im Parameter angegeben ist in das Set ein, sollte es nicht schon vorhanden sein.

boolean contains(Object obj)
Gibt true zurück, sofern sich das Objekt, welches im Parameter angegeben ist, in der HashSet befindet.

boolean remove(Object obj)
Löscht das Element, welches im Parameter angegeben ist, sofern es vorhanden ist.

bollean isEmpty()
Gibt true zurück, wenn sich kein Element im Set befindet.

void clear()
Löscht alle Elemente im Set.

int size()
Gibt die Anzahl der Elemente zurück, die sich im HashSet befindet.

Iteration

Die Iteration über ein HashSet kann mit einer einfachen foreach-Schleife vollbracht werden.

```
for (Object listObjects : setName)
        machWasMit(listObjects);
```

Hier als Beispiel, eine Methode die alle eingetragenen Nummern des Sets in der Konsole ausgibt:

```
public void allesAusgeben()
{
        for (Integer zahlen : zahlenMenge)
                System.out.print(zahlen + ", ");
}
```

Diese Methode kann wie bei der ArrayList verbessert werden.

Da die Methoden des HashSets sehr einfach und selbsterklärend sind, gibt es hier nur einfach einen Auszug aus meiner Klasse Hashset.

```
public void addZahl(int zahl)
{
    zahlenMenge.add(zahl);
}

public boolean zahlVorhanden(int zahl)
{
    return zahlenMenge.contains(zahl);
}

public void löschen(int zahl)
{
    if(zahlVorhanden(zahl) && !zahlenMenge.isEmpty())
        zahlenMenge.remove(zahl);
    else
        System.out.println("Diese Zahl ist nicht
vorhanden!");
}

public void allesLöschen()
{
    zahlenMenge.clear();
}

public int anzahlZahlen()
{
    return zahlenMenge.size();
}
```

Iterator

Der Iterator ist eine Klasse, die extra zum Iterieren über verschiedenste Sammlungen verwendet werden kann.

Der Iterator muss eigens importiert werden:

```
import java.util.Iterator;
```

Deklaration

```
Iterator< Object > itName = sammlungsName.iterator();
```

Der Iterator besitzt zwei Methoden:

boolean hasNext()
: Gibt true zurück, wenn sich noch Elemente in der Liste befinden.

Object next()
: Gibt das nächste Objekt der Sammlung aus, und schaltet den internen Zähler um eins weiter.

Mit diesen zwei Methoden kann eine schöne while-Schleife gebildet werden:

```
while (itName.hasNext())
    machWasMit(itName.next());
```

Zum Verständnis hier Methode die den Iterator verwendet. Hier wird die ArrayList namensListe verwendet.

```
public void ausgebenIterator()
{
   Iterator<String> iteratoOor = namensListe.iterator();
   while (iteratoOor.hasNext())
      System.out.print(iteratoOor.next() + ", ");
}
```

Der Iterator kann nicht bei der HashMap verwendet werden, da der Iterator ja nur einen Objekttyp halten kann, und die Map aus zwei verschiedenen Objekttypen besteht, nämlich dem Objekt für den Key und dem Objekt für die Value.