Collections Wissen

Einführung

Unter Collections sind verschiedene Möglichkeiten zusammengefasst, Daten zu strukturieren. Collection ist ein Interface in Java, das von verschiedenen Klassen implementiert wird, die zur Strukturierung von Daten auf verschiedene Art und Weise verwendet werden. Drei Interfaces erben vom Collection Interface:

- List

  Das Prinzip, das sich dahinter verbirgt, kennen wir bereits von Standardarrays. In einer Klasse, die List implementiert, werden Werte eines festgelegten Datentypen hintereinander abgelegt. Man startet häufig mit einer leeren List. Wenn wir Elemente hinzufügen, ändert sich die Größe dynamisch. Man kann durch einen Index auf die Elemente zugreifen (wie bei Arrays).
  Klassen, die dieses Interface implementieren: ArrayList, LinkedList, Vector
  Wir werden immer ArrayList verwenden, allerdings kommt man manchmal mit den anderen beiden Klassen in Berührung. Dann einfach denken, dass da ArrayList steht. Die Benutzung wird für uns gleich sein.
  Alle Klassen, die das List-Interface implementieren, können mit Collections.sort sortiert werden.

- Map

  Mit Maps werden assoziative Arrays verwaltet. Ein assoziatives Array speichert Werte unter einem Schlüssel von beliebigem Datentyp. Man kann sich das folgendermaßen vorstellen:
  Man hat einen großen Schlüsselbund mit verschiedenen Schlüsseln (keys einzigartig) und ist in einem Gebäude mit Türen (values). Mehrere Schlüssel können dieselbe Tür öffnen, aber ein Schlüssel kann nicht verschiedene Türen öffnen (ein key ist immer nur einem value zugeordnet). Es gibt bei Maps keinen Index.
  Klassen, die dieses Interface implementieren: HashMap, TreeMap
  Wir werden immer HashMap verwenden, allerdings kommt man manchmal mit der anderen Klasse in Berührung. Dann einfach denken, dass da HashMap steht. Die Benutzung wird für uns gleich sein.
  Wichtig: Maps werden sortiert, indem sie vorher in Listen umgewandelt werden.

- Set

  Mit Sets werden Mengen von Werten verwaltet. Man kann auf die einzelnen Werte aber nicht mit einem Index zugreifen wie bei Arrays. Man kann sich das vorstellen wie eine große Schale mit Kugeln, die alle unterschiedliche Farben haben. Es kann keine Kugel mit gleicher Farbe enthalten sein (einzigartige Werte). Wenn man eine Kugel herausnimmt, weiß man nicht, an welcher Stelle sie vorher war (kein Index). Es können nicht Kugeln und Würfel darin sein (nur von einem Datentyp).
  Klassen, die dieses Interface implementieren: HashSet, TreeSet
  Wir werden immer HashSet verwenden, allerdings kommt man manchmal mit der anderen Klasse in Berührung. Dann einfach denken, dass da HashSet steht. Die Benutzung wird für uns gleich sein.
  Wichtig: Sets werden sortiert, indem sie vorher in Listen umgewandelt werden.

Einführungsbeispiele (Stufe 1)

Beispiel für ArrayList:

```
ArrayList<String> al = new ArrayList<String>();

al.add("asdf");
al.add("asdfasdf");
al.add("asdfwg314g");
al.add("asdf");
al.add("asdg245h245");

System.out.println(al.get(3));

//-> asdf
```

Beispiel für HashMap:

```
//Zu jedem Gegenstand in meinem Zimmer (String) möchte ich einen Geldwert (double) zuordnen.

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

wertFuerGegenstaende.put("Schreibtisch", 10.0);
wertFuerGegenstaende.put("Bett", 400.0);
wertFuerGegenstaende.put("Regal", 80.0);
wertFuerGegenstaende.put("Spiegel", 20.0);


System.out.println(wertFuerGegenstaende.get("Bett"));

//-> 400.0
```

Beispiel für HashSet:

```
HashSet<Double> hs = new HashSet<Double>();

hs.add(8.0);
hs.add(84.0);
hs.add(-92.0);
hs.add(1.0);
hs.add(8.234);
hs.add(8.02);

//alle Elemente ausgeben
//die ersten drei Zeilen kann man auswendig lernen
Iterator<Double> it = hs.iterator();
while(it.hasNext()){
      double elem = it.next();
      System.out.println(elem);
}
```

Methoden (nur Namen)

ArrayList
- get
- add
- add (überladen)
- contains
- remove
- remove (überladen)
- size
- set
- iterator

HashMap
- get
- put
- containsKey
- containsValue
- remove
- keySet
- values
- size
- set
- iterator

HashSet
- add
- contains
- size
- remove
- iterator

und natürlich der jeweilige Konstruktor (siehe Beispiele Stufe 1)

Methoden (mit Datentypen und Erklärung)

Achtung! Die Methoden werden genau so benutzt, wie das immer funktioniert. Die Datentypen, die hier stehen, werden selbstverständlich nicht hingeschrieben. Sie stehen da trotzdem, damit man weiß, welche Rückgabedatentyp die Methoden haben und von welchem Datentyp die Parameter sein müssen.

ArrayList

#Datentyp der einzelnen Elemente# **get**(int)
Zum Auslesen: gibt ein Element der ArrayList abhängig vom Index (int) zurück.

void **add**(#Datentyp der einzelnen Elemente#)
Zum Speichern/Befüllen in die ArrayList: man übergibt ein neues Element, das an das Ende der ArrayList eingefügt wird (die Größe wird in diesem Moment um eins erhöht).

void **add**(int, #Datentyp der einzelnen Elemente#)
Zum Speichern/Befüllen in die ArrayList: man übergibt ein neues Element, das an der angegebenen Position (der 1. Parameter) in die ArrayList eingefügt wird (die Größe wird in diesem Moment um eins erhöht).

boolean **contains**(#Datentyp der einzelnen Elemente#)
Diese Methode überprüft, ob das übergebene Element in der ArrayList vorhanden ist.

void **remove**(int)
Diese Methode löscht das Element an der Stelle, die als Index übergeben wird. Die Elemente danach rutschen eins nach vorn. Die Größe der ArrayList verringert sich um eins.

void **remove**(#Datentyp der einzelnen Elemente#)
Diese Methode löscht das erste Vorkommen des übergebenen Elements. Die Elemente danach rutschen eins nach vorn. Die Größe der ArrayList verringert sich um eins.

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

al.add(1);
al.add(2);
al.add(3);
al.add(0, 4);

// Versucht, das Element an Index 5 zu löschen
// Wenn es nicht so viele Elemenete gibt, wird eine IndexOutOfBoundsException geworfen
al.remove(5);

// Versucht, das Element mit dem Wert 5 zu löschen
// Wenn es das Element nicht gibt, passiert gar nichts
al.remove(new Integer(5));
```

int **size**()
Liefert die Anzahl der Elemente zurück.

void **set**(int, #Datentyp der einzelnen Elemente#)
Diese Methode setzt das Element am angegeben Index auf den übergebenen Wert. Falls der Index zu groß ist, wird eine IndexOutOfBoundsException geworfen.

Iterator<#Datentyp der einzelnen Elemente#> **iterator**()
Mit dieser Methode bekommt man einen Iterator, den man dafür verwenden kann, alle Elemente nacheinander auszulesen (Empfehlung von Ronny: bei ArrayList stattdessen eine for-Schleife verwenden!)

#Datentyp der einzelnen Elemente#**[] toArray**(#Datentyp der einzelnen Elemente#**[]**)
Wenn an diese Methode ein Standardarray des Datentyps der einzelnen Elemente übergeben wird, kann man die ArrayList damit in ein Standardarray verwandeln, in dem die einzelnen Elemente denen aus der ArrayList entsprechen.

HashMap

#Datentyp der Values# **get**(#Datentyp der Keys#)
Zum Auslesen: Man übergibt einen Key und bekommt den passenden Value aus der HashMap. Falls der Key nicht enthalten ist, bekommt man null zurück.

void **put**(#Datentyp der Keys#, #Datentyp der Values#)
Zum Speichern/Befüllen in die HashMap: man übergibt den Key und den Value, die als Wertepaar in die HashMap hineingespeichert werden. Wenn der Key bereits vorhanden ist, wird der Value überschrieben, der vorher dem Key zugeordnet war.

boolean **containsKey**(#Datentyp der Keys#)
Diese Methode überprüft, ob der übergebene Key bereits in der HashMap enthalten ist.

boolean **containsValue**(#Datentyp der Values#)
Diese Methode überprüft, ob der übergebene Value bereits in der HashMap enthalten ist.

void **remove**(#Datentyp der Keys#)
Löscht ein Element der HashMap mit dem Key als Angabe. Die Größe verringert sich damit um eins. Wie man nach Values löscht, steht weiter unten.

Set<#Datentyp der Keys#> **keySet**()
Diese Methode gibt alle Keys als Set (siehe unten) zurück.

Collection<#Datentyp der Values#> **values**()
Diese Methode liefert alle Values der HashMap zurück. Wenn wir values() aufrufen, verwenden wir für gewöhnlich anschließend die Methode iterator() und scheren uns nicht um den für uns ungewöhnlichen Datentyp Collection.

int **size**()
Liefert die Anzahl der Elemente zurück

Iterator<#Datentyp der einzelnen Elemente#> **iterator**()
Mit dieser Methode bekommt man einen Iterator, den man dafür verwendet, die Keys oder die Values der HashMap alle durchzugehen (davor steht immer .keySet() oder .values())

HashSet

void **add**(#Datentyp der einzelnen Elemente#)
Zum Speichern/Befüllen: Der übergebene Wert wird in das HashSet gespeichert, wenn er noch nicht vorhanden ist. Ansonsten passiert nichts.

boolean **contains**(#Datentyp der einzelnen Elemente#)
Diese Methode überprüft, ob das übergebene Element in dem HashSet vorhanden ist

int **size**()
Liefert die Anzahl der Elemente zurück

void **remove**(#Datentyp der einzelnen Elemente#)
Löscht das übergebene Element

Iterator<#Datentyp der einzelnen Elemente#> **iterator**()
Mit dieser Methode bekommt man einen Iterator, den man dafür verwendet, alle Elemente nacheinander auszulesen

#Datentyp der einzelnen Elemente#**[] toArray**(#Datentyp der einzelnen Elemente#**[]**)
Wenn an diese Methode ein Standardarray des Datentyps der einzelnen Elemente übergeben wird, kann man das HashSet damit in ein Standardarray verwandeln, in dem die einzelnen Elemente denen aus dem HashSet entsprechen.

Beispiele (Stufe 2)

ArrayList

```
//Definition und Initialisierung
ArrayList<Double> al = new ArrayList<Double>();

System.out.println("Anzahl der Elemente: " + al.size());
//-> 0

//Bei Standardarrays legt man die Größe vorher fest und speichert dann die Elemente nach dem Index hinein. Hier ist es so, dass wir leer anfangen und dann nach und nach (meist in einer Schleife) Elemente hineinfüllen

al.add(8.01);
al.add(7.2);
al.add(7.2);
al.add(7.2);
al.add(9.6);

//Befüllen mit Schleife (das Befüllen der drei Collections (als Ziel) mit der Schleife ist abhängig von der Informationsquelle - hier ein Standardarray (deswegen for-Schleife) namens quelle)
double[] quelle = { 1.2, 3.1 };
for(int i = 0; i < quelle.length; ++i){
      al.add(quelle[i]);
}


//Setzen bzw. Überschreiben
//würden wir bei einem Standardarray so machen: arr[3] = 7.3;
al.set(3, 7.3);   //an Index Nummer 3 wird der Wert gespeichert

//Löschen
//Die ArrayList wird um je ein Element kleiner!
al.remove(al.size() - 1); //das letztes Element wird gelöscht
al.remove(0); //das erste Element wird gelöscht
//beim folgenden Aufruf ist das Element irgendwo; falls es mehrmals enthalten ist, wird das erste Vorkommen gelöscht
al.remove(7.2);
//falls es sich um eine Integer-ArrayList handelt, wird immer nach Index gelöscht!


//Sequentiell auslesen/verarbeiten (unsere ArrayList als Informationsquelle)
for(int i = 0; i < al.size(); ++i){
      System.out.println(al.get(i));
}
//->alle verbleibenden Elemente werden ausgegeben

//Achtung: diese beiden Methoden sind optional:
//Index von Elementen finden
System.out.println(al.indexOf(1.2));  //liefert den ersten Index des Elements zurück, wenn es mehrfach vorkommt
//-> 1
System.out.println(al.lastIndexOf(1.2));  //liefert den letzten Index des Elements zurück
//-> 1

//Herausfinden, ob ein Element überhaupt in der Collection ist
if(al.contains(-11.0))
      System.out.println("Yeah!");
else
      System.out.println("nein ...");
```

```
//–> nein ...
```

HashMap

```
//Definition und Initialisierung
HashMap<Double, Integer> hm = new HashMap<Double, Integer>();

//Wir fangen auch bei HashMap mit einer leeren Collection an und füllen sie später mit Werten

//Befüllen einzelner Elemente
//Achtung: wenn bei put ein Key angegeben wird, der bereits vorhanden ist, wird der Value dazu mit dem neuen Parameter überschrieben. Wenn put also zwei Mal mit dem gleichen Key aufgerufen wird, steht am Ende der Value vom zweiten Aufruf darin
hm.put(1.0, -10);
hm.put(6.0, 45);
hm.put(8.0, 11);
hm.put(16.0, 73);
hm.put(122.0, 334);
hm.put(1.9, 0);
hm.put(1.1, 0);

//Befüllen mit Schleife (das Befüllen der drei Collections (als Ziel) mit der Schleife ist abhängig von der Informationsquelle - hier ein Standardarray (deswegen for-Schleife) namens quelle)
double[] quelle = { 2.62, -89.8, 699.99, 11.1 };
for(int i = 0; i < quelle.length; ++i){
      hm.put(quelle[i], (int)quelle[i]);
}

//Setzen bzw. Überschreiben
System.out.println(hm.get(1.0));
//–> -10
hm.put(1.0, -9);
System.out.println(hm.get(1.0));
//–> -9

//Löschen
//nach key löschen
hm.remove(1.0);

//size() ist bei allen gleich
System.out.println("Anzahl der Elemente: " + hm.size());
//–> 10

//nach value löschen bzw. Löschen innerhalb von Iterator
//Hier im Beispiel werden alle Wertepaare gelöscht, bei denen der Value 0 ist
Iterator<Double> it = hm.keySet().iterator();
while(it.hasNext()){
      double currentKey = it.next();
     //Wenn die HashMap als Values Strings oder andere Referenztypen hat, verwenden wir equals, hier haben wir aber int, darum ==
      if(hm.get(currentKey) == 0){
            //löscht den aktuellen Wert aus der Collection
            //Wenn wir im Iterator sind, machen wir das immer genau so (das Löschen von Elementen)
            it.remove();
      }
}

System.out.println("Anzahl der Elemente: " + hm.size());
//–> 8

//Sequentiell auslesen/verarbeiten (unsere HashMap als Informationsquelle)
//nach Keys haben wir oben schon
```

```
//nach Values (keys sind uns gerade egal - wir wollen nur die Values verarbeiten):
Iterator<Integer> it = hm.values().iterator();
while(it.hasNext()){
	int currentValue = it.next();
	//value verarbeiten, zB ausgeben oder in eine ArrayList speichern oder oder oder ...
}


//Herausfinden, ob ein Element in der Map ist
if(hm.containsKey(1111.0))
	System.out.println("Yeah!");
if(hm.containsValue(45))
	System.out.println("Oh!");

//–> Oh!
```

HashSet

```
//Definition und Initialisierung
HashSet<Short> hs = new HashSet<Short>();


//Befüllen einzelner Elemente
//Achtung: wenn ein Wert bereits vorhanden ist, wird er nicht erneut hinzugefügt
hs.add(0);
hs.add(-2);
hs.add(6);
hs.add(18);
hs.add(132);

//Befüllen mit Schleife (das Befüllen der drei Collections (als Ziel) mit der Schleife ist abhängig von der Informationsquelle - hier ein Standardarray (deswegen for-Schleife) namens quelle)
short[] quelle = { 2, 3, 6, 132 };
for(int i = 0; i < quelle.length; ++i){
	hs.add(quelle[i]);
}

//size() ist bei allen gleich
System.out.println("Anzahl der Elemente: " + hs.size());
//–> 7

//Setzen bzw. Überschreiben
//Beim HashSet kann nichts überschrieben werden. Entweder etwas ist schon drin (dann passiert nichts - Wenn man die 3 schon drin hat und nochmal hs.add(3) aufruft, tut sich nichts) oder nicht (und das Element wird hineingespeichert). Man kann höchstens ein Element löschen, das man nicht mehr haben will (weiter unten)
hs.add(1);

//Löschen
//Achtung: hier wird kein Index übergeben, sondern direkt der Wert, der nicht mehr drin sein soll
hs.remove(1);

//Sequentiell auslesen/verarbeiten (unsere HashSet als Informationsquelle)
Iterator<Short> it = hs.iterator();
while(it.hasNext()){
	short elem = it.next();
	//verarbeiten ...
}

//Herausfinden, ob ein Element überhaupt in der Collection ist
if(hs.contains(-1000))
```

```
      System.out.println("Yeah!");
//-> (es wird nichts ausgegeben)
```

Beispiele (Stufe 3) toArray - Collection in Standardarray umwandeln

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

al.add(8.01);
al.add(7.2);
al.add(7.2);
al.add(7.2);
al.add(9.6);

//Achtung: Wrapperklasse muss verwendet werden

Double[] arr = al.toArray(new Double[0]);
for(int i = 0; i < arr.length; i++){
      System.out.println(arr[i]);
}
```

```
HashMap<Double, Integer> hm = new HashMap<Double, Integer>();

hm.put(1.0, -10);
hm.put(6.0, 45);
hm.put(8.0, 11);
hm.put(16.0, 73);
hm.put(122.0, 334);
hm.put(1.9, 0);
hm.put(1.1, 0);

//geht auch mit keySet, aber dann natürlich anderer Datentyp!
Integer[] arr = hm.values().toArray(new Integer[0]);
for(int i = 0; i < arr.length; i++){
      System.out.println(arr[i]);
}

//muss noch sortiert werden, falls notwendig
```

```
HashSet<Short> hs = new HashSet<Short>();

//Cast muss hier erfolgen - benutzt einfach kein short! Ist nur ein Beispiel
hs.add((short)0);
hs.add((short)-2);
hs.add((short)6);
hs.add((short)18);
hs.add((short)132);


Short[] arr = hs.toArray(new Short[0]);
for(int i = 0; i < arr.length; i++){
      System.out.println(arr[i]);
}
```

Beispiele (Stufe 4) asList - Standardarray in ArrayList umwandeln

```
Integer[] arr = { 2, 3 ,4 };
//Achtung: funktioniert nicht mit Standarddatentypen! Da müssen die Wrapperklassen genutzt werden

ArrayList<Integer> al = new ArrayList<Integer>(Arrays.asList(arr));

for(int i = 0; i < al.size(); i++){
	System.out.println(al.get(i));
}



//Noch interessanter: ArrayList in einer Zeile mit Elementen befüllen:

ArrayList<String> al = new ArrayList<String>(
	Arrays.asList(
		new String[]{ "what", "hey", "no" }));
```