Java

Generics & die Collections-API

Marcus Köhler

7. Dezember 2018

Java-Kurs

Überblick

1. Generics
   - Was sind Generics?
   - Wrapper-Klassen
2. Collections
   - Intro
   - Set und List
   - Iterieren
   - Map

Generics

Generics?

Was tun, wenn man Datenstrukturen wie Graphen und Listen auf einen bestimmten Typen beschränken will, aber nicht für jede denkbare Kombination aus Typen und Datenstruktur neuen Code schreiben will?

```
1 public class StringTree {
2     public void add(String item) {}
3     …
4 }
5 public class IntegerTree {
6     public void add(int item) {}
7      …
8 }
9 …
```

Generics?

Eine mögliche Antwort darauf ist Polymorphie. Allerdings hat man dabei das Problem, dass man schnell in komplexes casting rutscht, wenn man mehr als die Methoden des gemeinsamen „Vorfahren“ der zu verwendenden Typen benutzen will:

```
1 public class Tree {
2     public void add(Object item) {…}
3     public Object get(int id) {…}
4 }
5 …
6 Tree tree = new Tree();
7 tree.add("hello"); //works because any String is also an Object
8 String s = (String)tree.get(0); //must be cast because get(...) returns
      an Object
```

Generics?

Eine mögliche Antwort darauf ist Polymorphie. Allerdings hat man dabei das Problem, dass man schnell in komplexes casting rutscht, wenn man mehr als die Methoden des gemeinsamen „Vorfahren“ der zu verwendenden Typen benutzen will:

```
1 public class Tree {
2     public void add(Object item) {…}
3     public Object get(int id) {…}
4 }
5 …
6 Tree tree = new Tree();
7 tree.add("hello"); //works because any String is also an Object
8 String s = (String)tree.get(0); //must be cast because get(...) returns
      an Object
```

Hierbei passiert es schnell, dass man inkompatible Typen wie `String` und `int` vermischt und ein Fehler beim casting auftritt. Außerdem würde solcher Code ohne Fehler kompilieren, weshalb man nicht immer sofort bemerkt, wenn eine solche Situation auftritt.

Generics!

Um sogenannte *compile-time safety* zu garantieren, hat Java für solche Probleme die *Generics* eingeführt.
Mit Generics kann „automatisch“ der Typ von Attributen, Parametern und Rückgabewerten angepasst werden:

```
1 public class Box<T> {
2     // T stands for "Type"
3     private T t;
4
5     public void set(T t) { this.t = t; }
6     public T get() { return t; }
7 }
8
9 Box<Integer> integerBox; = new Box<Integer>();
```

Man kann auch mehr als ein Generic pro Klasse verwenden:

```
1 public class Dictionary<K,V> {
2     //K = Key, V = Value
3     ...
4     public V getByKey(K key) { ... };
5 }
```

Wrapper-Klassen

Primitive Datentypen können nicht als Typen von Generics auftreten. Um primitive Datentypen trotzdem zu verwenden, muss man die folgenden *Wrapper-Klassen* verwenden:

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

Collections

Die Collections-API

Java hat einige simple Datenstrukuren wie `Sets`, `Lists` und `Maps`. Diese Interfaces gehören zur sogenannten *Collections-API*.

Das *Package* `java.util` enthält neben den o.g. Interfaces auch verschiedene Implementierungen davon, wie z.B. `ArrayList`, `HashMap` etc. Alternativ kann man auch seine eigenen Implementierungen schreiben und verwenden.

java.util.Set

Ein `Set` aus dem `java.util`-Package ist eine Menge von Elementen des gleichen Typs, die kein Element zweimal enthalten kann.

```
1  import java.util.Set;
2  import java.util.HashSet;
3
4  public class SetTest {
5      public static void main(String[] args) {
6          Set<String> set = new HashSet<String>();
7
8          set.add("foo");
9          set.add("bar");
10         set.add("bar"); //throws IllegalMethodException
11         set.remove("foo");
12         System.out.println(set); // prints: [bar]
13     }
14 }
```

List

Eine `java.util.List` ist eine geordnete Menge von Objekten.
Eine `LinkedList` ist in Java als eine doubly-linked-list realisiert.

```
1  import java.util.List;
2  import java.util.LinkedList;
3
4  public static void main(String[] args) {
5      List<String> list = new LinkedList<String>();
6
7      list.add("foo");
8      list.add("foo"); // insert "foo" at the end
9      list.add("bar");
10     list.add("foo");
11     list.remove("foo"); // removes the first "foo"
12
13     System.out.println(list); // prints: [foo, bar, foo]
14 }
```

Listen-Methoden

Einige oft verwendete Methoden von `List<E>`:

void	`add(int index, E element)`	Element an Indexposition einfügen
E	`get(int index)`	Element an gegebenem Index ausgeben
E	`set(int index, E element)`	Element an Indexposition ersetzen
E	`remove(int index)`	Element an Indexposition entfernen

Spezifische Methoden von `LinkedList<E>`:

void	`addFirst(E element)`	Element am Anfang einfügen
E	`getFirst()`	Erstes Element ausgeben
void	`addLast(E element)`	Element am Ende einfügen
E	`getLast()`	Letztes Element ausgeben

for-each

Java hat eine spezielle Variante der `for`-Schleife für Collections, die `for-each`-Schleife:
`for (E e : collection)`

```java
public static void main(String[] args) {
    List<Integer> list = new LinkedList<Integer>();

    list.add(1);
    list.add(3);
    list.add(3);
    list.add(7);

    for (Integer i : list) {
        System.out.print(i + " "); // prints: 1 3 3 7
    }
}
```

„Hinter den Kulissen“ verwendet eine solche Schleife einen `Iterator`.

Iterator

Ein `Iterator` geht Element für Element über eine Collection:

```java
public static void main(String[] args) {
    List<Integer> list = new LinkedList<Integer>();

    list.add(1);
    list.add(3);
    list.add(3);
    list.add(7);

    Iterator<Integer> iter = list.iterator();
    while (iter.hasNext()) {
        System.out.print(iter.next());
    }
    // prints: 1337
}
```

Iterator

Ein „normaler“ `Iterator` hat die folgenden Methoden:

- `boolean hasNext()` - Gibt an, ob es noch Elemente nach dem momentanen Element gibt
- `E next()` - Gibt das nächste Element zurück(*kann nur einmal pro Element aufgerufen werden!*)
- `void remove()` - Entfernt das momentane Element(Wird nicht immer unterstützt)

Ein `Iterator` wird mittels `collection.iterator()` erzeugt:

```
1 Collection<E> collection = new Implementation<E>;
2 Iterator<E> iter = collection.iterator();
```

`Iterator`-Workflow

```
 Collection:  e1 e2 … eN
Iter.-Index:
```

1. Collection anlegen

`Iterator`-Workflow

```
 Collection:  e1 e2 … eN
Iter.-Index: ^
```

1. Collection anlegen
2. `Iterator` mittels `collection.iterator()` erzeugen

`Iterator`-Workflow

```
 Collection:  e1 e2 … eN
Iter.-Index:    ^
```

1. Collection anlegen
2. `Iterator` mittels `collection.iterator()` erzeugen
3. `Iterator.next()` aufrufen(gibt `e1` zurück);
   `Iterator.hasNext() == true`

`Iterator`-Workflow

```
 Collection:  e1 e2 … eN
Iter.-Index:      ^
```

1. Collection anlegen
2. `Iterator` mittels `collection.iterator()` erzeugen
3. `Iterator.next()` aufrufen(gibt `e1` zurück);
   `Iterator.hasNext() == true`
4. `Iterator.next()` aufrufen(gibt `e2` zurück);
   `Iterator.hasNext() == true`

`Iterator`-Workflow

```
 Collection:  e1 e2 … eN
Iter.-Index:        ^
```

1. Collection anlegen
2. `Iterator` mittels `collection.iterator()` erzeugen
3. `Iterator.next()` aufrufen(gibt `e1` zurück);
   `Iterator.hasNext() == true`
4. `Iterator.next()` aufrufen(gibt `e2` zurück);
   `Iterator.hasNext() == true`
5. …

`Iterator`-Workflow

```
 Collection:  e1 e2 … eN
Iter.-Index:           ^
```

1. Collection anlegen
2. `Iterator` mittels `collection.iterator()` erzeugen
3. `Iterator.next()` aufrufen(gibt `e1` zurück);
   `Iterator.hasNext() == true`
4. `Iterator.next()` aufrufen(gibt `e2` zurück);
   `Iterator.hasNext() == true`
5. …
6. `Iterator.next()` aufrufen(gibt `eN` zurück);
   `Iterator.hasNext() == false`

java.util.Map

Anders als die bisher vorgestellten Interfaces ist `java.util.Map` kein „Subinterface“ von `java.util.Collection`.
In einer `Map` sind Schlüssel und Werte gespeichert, wobei jeder Schlüssel einzigartig ist und jedem Schlüssel genau ein Wert zugeordnet ist. Werte können jedoch mehrfach vorkommen, weshalb zwei Schlüsseln der gleiche Wert zugeordnet sein kann.

```
1  public static void main (String[] args) {
2      Map<Integer, String> map = new HashMap<Integer, String>();
3
4      map.put(23, "foo");
5      map.put(28, "foo");
6      map.put(31, "bar");
7      map.put(23, "bar"); // "bar" replaces "foo" for key = 23
8
9      System.out.println(map);
10     // prints: {23=bar, 28=foo, 31=bar}
11 }
```

keySet() & values()

Da die Schlüssel einer `Map` einzigartig sind, kann man sie mittels `keySet()` als `Set` repräsentieren. Das gleiche gilt *nicht* für die zugeordneten Werte: da diese mehrmals auftreten können, werden sie von der `values()`-Methode als Collection zurückgegeben.

```
1  public static void main (String[] args) {
2      // [...] map like previous slide
3
4      Set<Integer> keys = map.keySet();
5      Collection<String> values = map.values();
6
7      System.out.println(keys);
8      // prints: [23, 28, 31]
9
10     System.out.println(values);
11     // prints: [bar, foo, bar]
12 }
```

Map.keySet().iterator()

Um mit einem `Iterator` über eine `Map` zu iterieren, kann man mit dem `Iterator` des Schlüssel-Sets arbeiten:

```
1  public static void main (String[] args) {
2
3      // [...] map, values like previous slide
4      Set<Integer> keys = map.keySet();
5      Iterator<Integer> iter = keys.iterator();
6
7      while(iter.hasNext()) {
8          System.out.print(map.get(iter.next()) + " ");
9      } // prints: bar foo bar
10
11     System.out.println(); // print a line break
12
13     for(Integer i: keys) {
14         System.out.print(map.get(i) + " ");
15     } // prints: bar foo bar
16 }
```

Map und for-each

Man kann auch mit einem `for-each` direkt über dem Keyset iterieren und dadurch über die `Map` iterieren:

```
1  Map<String, String> map = ...
2  for (Map.Entry<String, String> entry : map.entrySet()) {
3    System.out.println("Key: " + entry.getKey() +
4    ", value" + entry.getValue());
5  }
```

Zusammenfassung

List	Speichert die Reihenfolge der Elemente Einfach zu iterieren ineffektive Suche
Set	Keine Duplikate Keine bestimmte Reihenfolge Effektives Suchen
Map	Key-Value=Paare Hocheffektive Suche Iteration komplizierter

Übung

Wähle Dir eine der folgenden Übungen aus:

1. (Generics) Baue die `NumberBox` aus dem Kurs-Repo(unter sourcefiles/07-collections/exercise) so um, dass sie Generics verwendet.

2. (Collections) Schreibe ein Programm, in dem du eine Implementierung von `java.util.List` mit Elementen befüllst und diese mittels eines `Iterators` ausgibst. Welche Methoden werden von diesem `Iterator` verwendet?