Verwendung von Collections & Lambda-Funktionen

Lernziele:

- Sie kennen den Aufbau der Collections
- Sie kennen die *Map*-Datenstruktur und können es anwenden
- Sie können die Bedeutung von generischen Klassen und Schnittstellen erklären
- Sie können Lambda-Funktionen für Suche und Sortieren innerhalb einer Collection anwenden

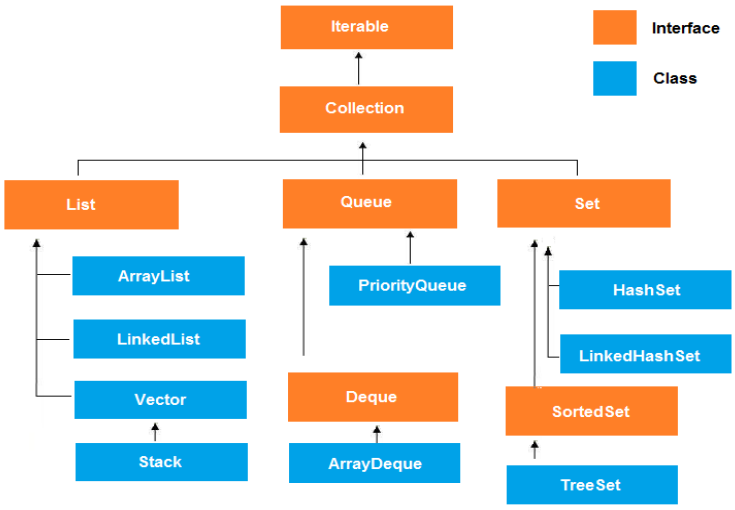
1 Collection Klassen mit List

Das Package java.util stellt häufig genutzte Datenstrukturen wie Listen, Stacks und Queues mit wichtigen Funktionen bereit. Durch den Einsatz dieser Klassen können sich Entwickler viel Zeit zum Erstellen und Testen solcher Datenstrukturen sparen und sich auf die eigentliche Problemlösung konzentrieren.

1.1 Die Schnittstellen Collection und List

Aus dem Modul 226 kennen Sie Datenstrukturen von java.util, d.h. von der gemeinsamen Schnittstelle ”Collection”. Diese Schnittstelle (oder *Interface*) definiert allgemeine Methoden und Funktionen, die innerhalb der Klassen implementiert sind. Eine Schnittstelle selbst beinhaltet allerdings keinen Code.

Abbildung 1: Quelle: framework.html



Dadurch wissen Entwickler bei der Benutzung einer Klasse, die die Schnittstelle implementiert, auf welche Funktionen sie zugreifen können. Dies dient der schnelleren und stabileren Software-Entwicklung.

Beispiele für Methoden des Collection Interfaces sind:

- `add(Object o)`
- `remove(Object o)`
- `iterator()`
- `size()`
- `stream()`
- `toArray()`

Das *List* – Interface definiert zusätzliche Methoden, um über Index-Adressen auf Listenelemente zuzugreifen:

- `remove(int pos)`
- `add(<T> object, int pos)`

Mit dem ebenfalls definierten *ListIterator* können die Datenstrukturen vorwärts UND rückwärts durchschritten werden.

1.2 Implementierung der Schnittstellen in Klassen des java.util Pakets

Die bisher behandelten Datenstrukturen sind in Java auch in der Collections-Bibliothek aufgeführt.

verwendete Datenstruktur	***In Collections-Bibliothek***
primitiver Array	*ArrayList** oder *Arrays*
Verkettete Liste	*LinkedList**
Eigener Stack / Stapel	*Stack*
noch nicht behandelt	*Map*

*beide Klassen implementieren das Interface *List*

Wie oben aufgeführt, ist der *Stack* eine Erweiterung der *Vector* Klasse. Der *Vector* wiederum ist eine spezielle Form eines Arrays. Wie bei *ArrayList* ist ein *Vector* dynamisch erweiterbar, hat aber den Vorteil, dass es **synchronized** ist (d.h. er kann zur Laufzeit nicht von einem anderen Prozess geändert werden).

Die Verwendung des Interfaces Collection ermöglicht, dass wir bei der Instanziierung eines Objekts den konkreten Datentyp der Elemente definieren können:

```
List<String> myStrings = new ArrayList<String>();
Stack<Card> deck42 = new Stack<Card>();
```

Solche Klassen und Schnittstellen werden in Java und C# ”generisch” bezeichnet. In C++ wird der Begriff Template-Klasse genutzt.

1.3 Die *foreach*-Schleife mit Collections

Eine grundlegende Operation auf Collections ist die Verwendung von *foreach*, mit der wir alle Elemente in einer Collection durchlaufen und bearbeiten können. Das führt zu sehr kompaktem Code, wie in diesem Beispiel:

Beispiel für einen Stack:

```
Stack<String> collection = new Stack<String>();
…
for (String s: collection){
      System.out.println(„Element = „ + s);
}
```

Welche andere *for*-Schleife könnte man für die gleiche Operation verwenden?
Notieren Sie diese andere *for*-Schleife (diese sollten sie jetzt ziemlich gut kennen ☺)

Beide Schleifen (*foreach* und das normale *for*) sind eine Kurzform für ein *while*-Konstrukt. Wen man das Interface *Iterator* verwendet, kann man eine sehr elegante while-Schleife verwenden. Wir haben das schon bei der Erweiterung unseres Stacks angetroffen:

```
Iterator<String> i = collection.iterator();
while (i.hasNext()) {
      String s = i.next();
      System.out.println(s);
}
```

Der *Iterator* selbst ist ein Interface. Sobald eine Klasse dieses Interface implementiert, hat es automatisch das Verhalten eines Iterators. Siehe dazu `java.util.Iterator`.

2 Lambda-Funktionen zum Suchen und Sortieren

In der Praxis werden Listen häufig nach bestimmten Kriterien sortiert. Mit der Methode *sort*(..) stellt *ArrayList* ein leistungsfähiges Werkzeug dazu bereit. *sort*(..) bekommt als Parameter eine sogenannte Lambda-Funktion übergeben.

Lambda-Funktionen stammen aus der Welt der funktionalen Programmierung (LISP). Sie wurden mit Java SE8 zur Sprachspezifikation hinzugefügt und ermöglichen sehr elegantes Sortieren und Suchen.
Dabei kann man sich Lambda-Ausdrücke einfach als ein Stück Code vorstellen, das zu einem Zeitpunkt ausgeführt wird. Dieses Stück Code braucht selber keine Methode.

Beispiel:

```
Integer.compare(first.length(), second.length());
```

Hier werden zwei String-Objekte verglichen.

Als Lambda-Ausdruck:

```
(String first, String second) -> Integer.compare(first.length(),
second.length());
```

Der Ausdruck ist ein Stück Code, mit einer Beschreibung der Variablen, die benötigt werden. Die Zeichenfolge ” -> ” stellt die Zuordnung der Variablen zur Funktion dar.

Wir wollen hier zwei Beispiele genauer ansehen:

Die Klasse *Person* enthält die Attribute *familyName, firstName, city* und *birthDate* als Strings. In der ArrayList *kundenverwaltung* sind alle Kunden der Topomedics AG instanziert. Nun kann die Liste mit der *sort*() Methode beispielsweise nach Namen sortiert werden:

```
ArrayList<Person> kundenverwaltung = loadAllData();
kundenverwaltung.sort((p1,p2)->p1.familyName.compareTo(p2.familyName));
```

Die Methode *void sort(...)* übergibt als Parameter einen Lambda-Ausdruck, der das Sortierkriterium beschreibt. Dabei sind p1, p2 die Variablennamen der jeweils zu vergleichenden Listenelemente für die interne Sortierfunktion. Entsprechend sind p1 und p2 vom Datentyp *Person*. Die Namen p1, p2 können frei gewählt werden. Wir greifen also auf das Attribut *familyname* von p1 zu. Dieses ist vom Datentyp *String* und ermöglicht somit die Anwendung der String – Methode *compare*. *Compare* bekommt das *familyName* – Attribut von p2 als Parameter. *String.compare* gibt -1 zurück, wenn p1 im Alphabet vor p2 kommt bzw. 1 wenn es nach p2 im Alphabet kommt.

Da *sort*() keinen Rückgabetyp hat, wird *kundenverwaltung* in sich selbst sortiert. Auf

Englisch heisst das ”in-place”.

2.1 Teillisten mit Lambda-Ausdrücken extrahieren

Häufig werden aus Listen bestimmte Teilmengen benötigt. Nun sollen wir beispielsweise aus der *kundenverwaltung* nur die Personen finden, die aus Appenzell stammen. Dazu können wir entweder klassisch alle Elemente der *kundenverwaltung* Liste einzeln aufrufen und das city – Attribut auf ”Appenzell” vergleichen. Eleganter ist folgende Syntax:

```
List<Person> appenzeller = kundenverwaltung.stream().
filter(p->p.city.equals("Appenzell")).//filter by city!
collect(Collectors.toList());
```

Hier werden die Elemente der Liste *kundenverwaltung* mit der stream() Methode an die Methode *filter*(..) übergeben. *filter(p->p.city.equals(”Appenzell”))* enthält wieder einen Lambda-Ausdruck. Zunächst wird die Variable p definiert. p ist vom Typ *Person* und hat ein Klassenattribut namens *city* vom Datentyp *String*.
Ein String x wird mittels x.equals(String y) – Funktion auf Gleichheit geprüft. Dieser Ausdruck wird wahr, wenn alle Zeichen in x in der gleichen Reihenfolge in y vorkommen. Der *filter*(..) Aufruf gibt also nur die Instanzen von Person weiter, die aus ”Appenzell” stammen. Die Methode *collect(Collectors.toList())* sammelt nun die gefilterten Personen und hängt diese fein säuberlich in eine neue Liste. Diese speichern wir in der neuen Variable List<person> *appenzeller* ab.

Entsprechend kann beispielsweise eine Zahlenliste nach Zahlen grösser 10 und kleiner 20 wie folgt gefiltert werden:

```
List<Integer> kleiner10 = zahlen.stream().
filter(z -> z>10 && z<20).
collect(Collectors.toList());
```

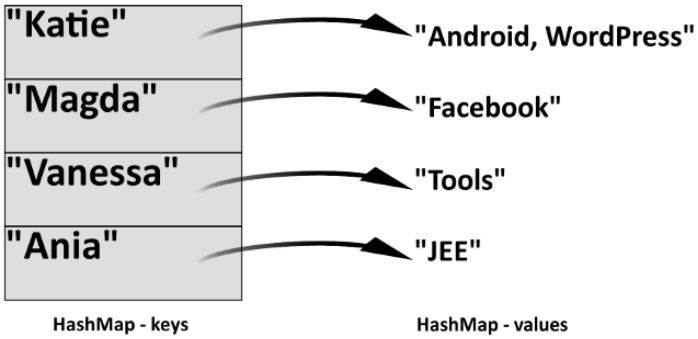
Probieren Sie es aus!

3 Die *Map* Datenstruktur

Der Hauptzweck der *Map* Datenstruktur ist es, *Werte* (*values*) mit *Schlüsseln* (*key*) zu verbinden. Sie unterstützt üblicherweise 2 Operationen:

- *einfügen* (*put*) eines neuen Paares
- *suchen* (*get*) nach dem Wert, der mit einem gegebenen Schlüssel verbunden ist

Diese Struktur wird sehr oft eingesetzt. Ein klassisches Beispiel ist das Telephonbuch (oder auch Kontakte auf Ihrem Smartphone).



Dabei muss darauf geachtet werden, dass es nicht zu doppelten Einträgen (doppelte Schlüssel) kommt. Somit wird folgende Regel implementiert:

- Jeder Schlüssel ist immer nur mit einem Wert verbunden
- Wenn ein Client ein Schlüssel-Wert-Paar einfügt, in der dieser Schlüssel (und ein damit verbundener Wert) bereits vorhanden ist, dann ersetzt der neue Wert den alten.

In Java wird das Interface *Map* in vielen verschiedenen Klassen implementiert (siehe dazu Java API). Eine sehr oft verwendete Klasse ist der *HashMap*.

Code-Beispiel:

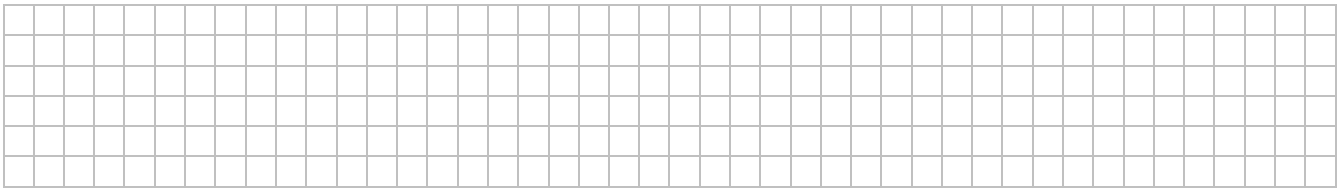
```
HashMap<String, String> phoneBook =    new HashMap<String, String>();

phoneBook.put(„Lisa Jones“, „(402) 4536 4674“);
phoneBook.put(„Prince Harry“, „(0044)79854 4512“);
```

3.1 Aufgabe: *HashMap* kennenlernen

Schreiben Sie eine Klasse, welche ein *HashMap* benützt, um Telefon-Einträge zu verwalten. Dabei soll die Tel.Nummer den *Key* bilden und der Name den *Value*.

Der Benutzer kann via Konsole / GUI einen Kontakt eintragen, der in dieser *HashMap* abgelegt wird.

Mittels `get(Object key)` erhalten wir den *Value*. Wie kann über das ganze Map iteriert werden?



3.2 Aufgabe: *HashMap* für grössere Datenmengen verwenden

Wir wollen die *HashMap* für eine Liste von US-Flughäfen verwenden (siehe airports.cvs).
Schreiben Sie eine Klasse, welche diese Liste in ein HashMap einliest.
Erweitern Sie die Klasse, so dass nach einem Flughafen Code gesucht werden kann.
(entweder via Konsole oder GUI)

3.3 Für Fortgeschrittene: Tech-Support System implementieren

Erstellen Sie ein Tech-Support System, welches anhand von bestimmten Schlüsselwörtern, dem User eine „Hilfe" anbietet. Diese Applikation sollte folgende Klassen beinhalten:

InputReader: diese Klasse benützt die *Scanner*-Klasse, um die Eingabe einzulesen.

Responder: diese Klasse hat ein *HashMap* mit Schlüsselwörtern (*key*) und entsprechenden Antworten (*value*).

SupportSystem: hier wird Eingabe und Ausgabe gesteuert.
Diese Klasse benützt den *InputReader* für die Eingabe und den *Responder* für eine Antwort.

Ein Auszug der Klasse mit der Methode *start()*:

```
public class SupportSystem
{
    private InputReader reader;
    private Responder responder;

    …

    public void start()
    {
        boolean finished = false;

        printWelcome();

        while(!finished) {
            String input = reader.getInput();

            if(input.startsWith("bye")) {
                finished = true;
            }
            else {
                String response = responder.generateResponse();
                System.out.println(response);
            }
        }
        printGoodbye();
    }
```

3.4 Zusatz-Aufgabe: Elemente entfernen

Auch bei einer HashMap verwenden wir den *Iterator*, um über die einzelnen Elmente zu iterieren (d.h. sie einzeln durchgehen).

```
Iterator myIterator = myHash.entrySet().iterator();

while (myIterator.hasNext()){

Map.Entry element = (Entry) myIterator.next();
System.out.println("key = " + element.getKey() + " = " + element.getValue());

}
```

Es ist aber auch mit einem Lambda-Ausdruck möglich:

```
myHash.forEach((k,v) -> System.out.println("Key: " + k + ": Value: " + v));
```

Wie können Sie Elemente aus einer HashMap entfernen?