Exercises – Law Of The Big Three

PART 1 + 2 + 3

Aufgabe 1) Die Klasse `Order` 5 min

Korrigiere bzw. vervollständige die Klasse `Order` so, dass die Speicherung in Listen und `HashSet` einen erwartungskonformen Inhalt besitzen, also Elemente mit gleicher Id und gleichem Namen nicht mehrmals im Set auftreten.

Aufgabe 2) Korrigiere die Klasse `DepartueTable` mit (verschachtelten) Arrays 10 min

Implementiere die Vergleiche erstmal von Hand bzw. begutachte die bestehende Implementierung. Analysiere das Problem und korrigiere es von Hand – bevor ggf. die Automatiken der IDE zum Einsatz kommen.

```
public static void main(String[] args)
{
     var dt1 = new DepartureTable();
     var dt2 = new DepartureTable();

     System.out.println(dt1.equals(dt2));
}
```

TIPP: Schau dich in der Klasse `Arrays` um!

BONUS: Was wäre ggf. eine adäquatere Art der Datenspeicherung? 10 min

Aufgabe 3) Korrigiere die Klasse `Person` 10 min

Für die Klasse `Person` wurden die Methoden `equals()` und `hashCode()` bereits passend bereitgestellt. Wieso scheint die Funktionalität aber doch nicht korrekt gegeben? Korrigiere das Problem des Wiederfindens nach Änderungen, hier etwa, wenn eine Person umzieht.

---------- ca. 35 min Übungen

PART 4

Aufgabe 4) Korrigiere die Klasse `Customer` 10 min

Analysiere warum es zu Abweichungen kommt, wenn man Objekte der Klasse `Customer` in einem `HashSet` oder einem `TreeSet` speichert. Korrigiere das.

BONUS: Definiere passende Komparatoren und verbinde diese geeignet, sodass eine verständliche und lesbare Implementierung entsteht. **10 min**

Aufgabe 5) Kennenlernen der Klasse `LinkedHashMap` 20 min

Die Klasse `LinkedHashMap` merkt sich die Einfügereihenfolge, was für diverse Anwendungsfälle sehr praktisch ist. Das gilt insbesondere, weil der performante Zugriff der Klasse `HashMap` beibehalten wird.

Die Klasse `LinkedHashMap` bietet zwei Besonderheiten, nämlich die Möglichkeit einer Größenbeschränkung und zudem noch die Beachtung einer Zugriffsreihenfolge.

```
// Initial befüllen
fixedSizeMap.put("Erster",
                 new SimpleCustomer("Erster", "Stuhr", 11));
fixedSizeMap.put("Zweiter",
                 new SimpleCustomer("Zweiter", "Hamburg", 22));
fixedSizeMap.put("M. Inden",
                 new SimpleCustomer("Inden", "Aachen", 39));
printCustomerList("Initial", fixedSizeMap.values());

// Änderungen durchführen und ausgeben
fixedSizeMap.put("New1", new SimpleCustomer("New_1", "London", 44));
printCustomerList("After insertion of 'New_1'",
fixedSizeMap.values());

fixedSizeMap.put("New2", new SimpleCustomer("New_2", "San
Francisco", 55));
printCustomerList("After insertion of 'New_2'",
fixedSizeMap.values());
```

Diese Abfolge sollte diese Ausgaben produzieren:

```
Initial
Customer [name=Erster, city=Stuhr, age=11]
Customer [name=Zweiter, city=Hamburg, age=22]
Customer [name=Inden, city=Aachen, age=39]
```

```
After insertion of 'New_1'
Customer [name=Zweiter, city=Hamburg, age=22]
Customer [name=Inden, city=Aachen, age=39]
Customer [name=New_1, city=London, age=44]
After insertion of 'New_2'
Customer [name=Inden, city=Aachen, age=39]
Customer [name=New_1, city=London, age=44]
Customer [name=New_2, city=San Francisco, age=55]
```

BONUS: Es sollen auch noch die Zugriffe beachtet werden, damit nicht normal nachgeschoben wird, sondern unter Beachtung der letzten Zugriffe.
Damit lässt sich auf einfache Weise ein LRU-Cache selbst realisieren.

---------- ca. 40 min Übungen

Aufgabe 6 BONUS) Check Magic Triangle 30 min

Schreiben Sie eine Methode `boolean isMagicTriangle(List<Integer> values)`, die prüft, ob eine Folge von Zahlen ein magisches Dreieck bildet. Ein solches ist definiert als ein Dreieck, bei dem die jeweiligen Summen der Werte der drei Seiten alle gleich sein müssen. Beginnen Sie mit einer Vereinfachung für die Seitenlänge 3 und einer Methode `boolean isMagic6(List<Integer> values)`.

Beispiele Nachfolgend ist das für je ein Dreieck der Seitenlänge drei und der Seitenlänge vier gezeigt:

```
  1          2
 6 5        8 5
2 4 3      4   9
          3 7 6 1
```

Damit ergeben sich folgende Seiten und Summen:

Eingabe	Werte 1	Werte 2
Seite 1	1 + 5 + 3 = 9	2 + 5 + 9 + 1 = 17
Seite 2	3 + 4 + 2 = 9	1 + 6 + 7 + 3 = 17
Seite 3	2 + 6 + 1 = 9	3 + 4 + 8 + 2 = 17

Tipp: Extrahieren Sie die Seiten mithilfe von `subList(startIdx, endIdx)`.

PART 5: Iteratoren

Aufgabe 7) Every2nd-Iterator 15 min

In dieser Aufgabe soll ein einfacher Iterator implementiert werden, der ausgehend von der Startposition, dann jeweils das 2. Element einer Datenstruktur ausgibt.

Für die Ausgangsdaten

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

erwarten wir folgendes Ergebnis: [1, 3, 5, 7, 9]

Aufgabe 8) EveryNth-Iterator 15 min

Implementieren Sie einen Iterator namens `EveryNth`, der jedes n-te Element durchläuft. Also für die Eingabe `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]` beispielsweise folgende Werte ausgibt:

- Mit Schrittweite 3: `1, 4, 7 ,10`
- Mit Schrittweite 5: `1, 6, 11`

---------- ca. 30 min Übungen

Weiterführende Links

- https://www.javacodegeeks.com/2019/05/java-equals-hashcode.html
- https://www.baeldung.com/java-equals-hashcode-contracts
- https://examples.javacodegeeks.com/java-hashcode-method-example/

Fragestunde

Mittlerweile solltest du folgende Fragen beantworten können:

1. What is the difference between "==" and equals() method in Java?

2. Can two objects which are not equal have the same hashCode?

3. How does get() method of HashMap works, if two keys have the same hashCode?

4. What happens if equals() is not consistent with compareTo() method?

Antworten

1. What is the difference between "==" and equals() method in Java?

- == => Refrenzvergleich
- equals() => inhaltlicher / semantischer Vergleich (auf Basis der relevanten Attribute)

2. Can two objects which are not equal have the same hashCode?

YES, two objects, which are not equal to `equals()` method can still return same `hashCode`.

3. How does get() method of HashMap works, if two keys have the same hashCode?

When two keys return same `hashcode`, they end up in the same bucket. Now, in order to find the correct value, you used `equals()` method to compare with key stored in each `Entry` of linked list there.

4. What happens if equals() is not consistent with compareTo() method?

Some `java.util.Set` implementation e.g. `SortedSet` or it's concrete implementation `TreeSet` uses `compareTo()` method for comparing objects.
If `compareTo()` is not consistent means doesn't return zero, if `equals()` method returns `true`, it may break `Set` contract, which is not to avoid any duplicates.