CODEGYM

Mentor-supported training program from CodeGym

Java Developer in 12 months

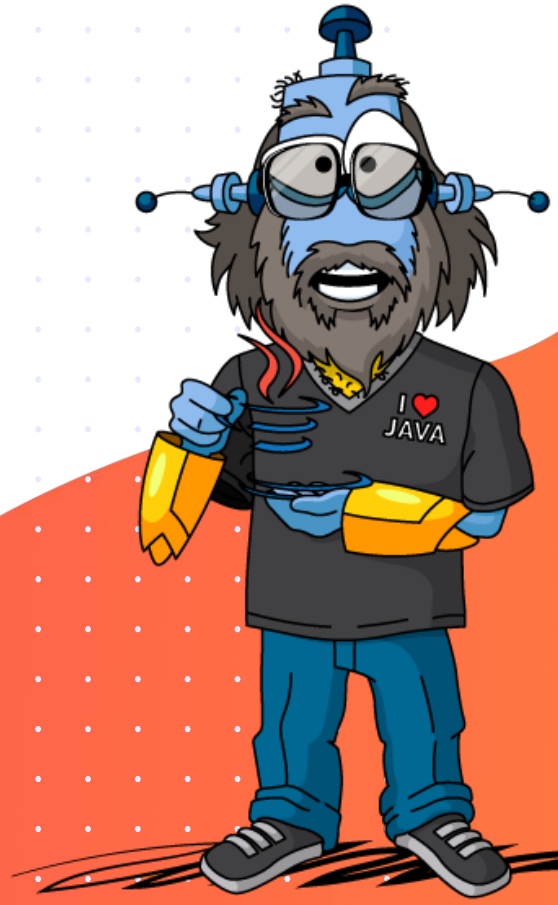
MODULE 1. JAVA SYNTAX

Lesson 17 Collections, part 2

CODEGYM

Lesson plan

- Introducing the HashMap collection
- Introducing the TreeMap collection
- Tips on using collections
- Collections — a helper class



CODEGYM

Map

In Java, another interesting collection (broadly speaking) is Map.

The data type is officially known as an "**associative array**", but that term is rarely used. A more common alternative name is "**dictionary**".

Inside a Map, data is stored in key-value pairs.

The keys and values can be any objects — numbers, strings, or objects of other classes.

CODEGYM

HashMap

The HashMap class is the most popular kind of Map collection. On the one hand, it is very similar to HashSet and has all its methods. On the other hand, it is like a list (ArrayList) that can use objects as its indices.

You can create a HashMap using a statement like this:

```
HashMap<KeyType, ValueType> name = new HashMap<KeyType, ValueType>();
```

Where **KeyType** is the type of the keys in the stored pairs, and **ValueType** is the type of the values in the pairs stored in the HashMap collection.

CODEGYM

Basic methods of HashMap

Method	Description
`void put(KeyType key, ValueType value)`	Adds the (key, value) pair to the collection
`ValueType get(KeyType key)`	Returns the value associated with a key.
`boolean containsKey(KeyType key)`	Checks whether a key exists in the collection
`boolean containsValue(ValueType value)`	Checks for the existence of a value in the collection
`ValueType remove(KeyValue key)`	Removes a pair from the collection
`void clear()`	Clears the collection, removing all the elements
`int size()`	Returns the number of key-value pairs in the collection
`Set<KeyType> keySet()`	Returns the set of keys in the collection
`Collection<ValueType> values()`	Returns a set containing the elements of the collection
`Set<Map.Entry<KeyType, ValueType>> entrySet()`	Returns a set (Set) of all pairs (Map.Entry) in the collection.

CODEGYM

Subsets of a HashMap: the set of keys

There are many ways to get a list of all the keys and values. The easiest way is to loop over the keys.

HashMap entries are not numbered sequentially, so a loop with a counter won't work here. But we can get a set of keys using the keySet() method

```
Set<KeyValue> keys = map.keySet();
```

There is also a more complicated way: you can transform a Map into a set of key-value pairs, and then loop over the elements of the set.

The result of calling the **entrySet()** method on a **HashMap<KeyType, ValueType>** object will be a **Set<Entry<KeyType, ValueType>>**:

```
Set<Entry<KeyType, ValueType>> name = map.entrySet();
```

CODEGYM

Introducing the TreeMap collection

TreeMap is a dictionary-like data structure that lets you store key-value pairs sorted by key.

A comparator is used to compare keys. Many JDK classes implement the Comparable interface, which just indicates that objects of this type can be sorted. The default sort uses natural ordering.

A class does not have to implement the Comparable interface in order to be used as the key in a TreeMap. But when creating a TreeMap object, you need to specify the rule that will be used to sort the keys.

CODEGYM

Example of an anonymous comparator in TreeMap

```
public static void main(String[] args) {
    //Student ranking
    var students = new TreeMap<Student, Boolean>(new Comparator<Student>() {
        @Override
        public int compare(Student o1, Student o2) {
            int gradeFirst = o1.engineering + o1.english + o1.math;
            int gradeSecond = o2.engineering + o2.english + o2.math;
            return Integer.compare(gradeFirst, gradeSecond);
        }
    });
}

class Student {
    //Academic subjects
    int engineering;
    int math;
    int english;
}
```

A TreeMap can have a null key, but to avoid getting a NullPointerException, you must explicitly set a comparator that knows how to deal with null

CODEGYM

Tips on using collections

If we need a List, then we use the ArrayList implementation

List is ideal for storing a set that may contain duplicate elements, may change in size, and needs to be accessible by index.

If we need a Set, then we use the HashSet implementation

Set is ideal for storing unique elements without worrying about their order.

If we need a Map, then we use the HashMap implementation

Map is ideal when you need to store the mapping of one object to another.

Choose the correct collection for the task at hand.

	By index	By key	Search	Insert at the end	Insert in the middle	Delete
ArrayList	O(1)	N/A	O(n)	O(1)	O(n)	O(n)
LinkedList	O(n)	N/A	O(n)	O(1)	O(1)	O(1)
HashSet	N/A	O(1)	O(1)	N/A	O(1)	O(1)
TreeSet	N/A	O(1)	O(log n)	N/A	O(log n)	O(log n)
HashMap	N/A	O(1)	O(1)	N/A	O(1)	O(1)
TreeMap	N/A	O(1)	O(log n)	N/A	O(log n)	O(log n)

CODEGYM

Creating and modifying collections

`Collections.addAll(Collection<T> colls, T e1, T e2, T e3, ...)` method
The addAll() method adds the elements e1, e2, e3, ... to the colls collection Any number of elements can be passed.

`Collections.fill(List<T> list, T obj)` method
The fill() method replaces all the elements of the list collection with the obj element.

`Collections.nCopies(int n, T obj)` method
The nCopies() method returns a list of n copies of the obj element. Note that the returned list is immutable, which means you cannot change it!
You can only use it to read values.

`Collections.replaceAll (List<T> list, T oldValue, T newValue)` method
The replaceAll() method replaces all the elements in the list collection equal to oldValue with newValue.

`Collections.copy (List<T> dest, List<T> src)` method
The copy() method copies all the elements of the src collection into the dest collection.

CODEGYM

Finding elements in collections

`Collections.min(Collection<T> colls)` method
The `min()` method returns the minimum element in the collection.

`Collections.max(Collection<T> colls)` method
The `max()` method returns the maximum element in the collection.

`Collections.frequency(Collection<T> colls, T obj)` method
The `frequency()` method counts the number of times the `obj` element occurs in the colls collection

`Collections.binarySearch(Collection<T> colls, T obj)` method
The `binarySearch()` method searches for the `obj` element in the `colls` collection. Returns the index of the found element. Returns a negative number if the element is not found.

`Collections.disjoint(Collection<T> coll1, Collection<T> coll2)` method
The `disjoint()` method returns true if the passed collections do not have any elements in common.

CODEGYM

Order of the elements

`Collections.reverse(List<T> list)` method
The `reverse()` method reverses the order of the elements of the passed list.

`Collections.sort(List<T> list)` method
The `sort()` method sorts the passed list in ascending order.

`Collections.rotate(List<T> list, int distance)` method
The `rotate()` method cyclically shifts the elements of the passed list by distance positions forward.

`Collections.shuffle(List<T> list)` method
The `shuffle()` method randomly shuffles all the elements of the passed list. The result is different every time.

CODEGYM

Homework

MODULE 1. JAVA SYNTAX

Complete Level 18

CODEGYM

Answers to questions

MODULE 1. JAVA SYNTAX