Module 1-8

Collections: Maps and Sets

Maps: Introduction

Maps are used to store key-value pairs.

- Examples of key value pairs: dictionary entries (word => definition), a phone book (name => phone number), a list of employees (employee number => employee name)
- We will focus on the most common type of map, the HashMap.

Maps: Declaring

Maps follow this declaration pattern:

```
import java.util.HashMap;
import java.util.Map;

public class MyClass {

        public static void main(String args[ ]) {

                Map <Integer, String> myMap =
                                new HashMap<Integer, String>();

        }
}
```

We will need these 2 imports for a hash map.

We are creating a type of Map called a HashMap

We have specified that the key will be an integer and the value will be the String

Note the “**new**” keyword which **instantiates** the map.

Maps: put method

The put method adds an item to the map. The data types must match the declaration.

```
Map <Integer, String> myMap = new HashMap<>();
myMap.put(1, "Rick");
myMap.put(2, "Beth");
myMap.put(3, "Jerry");
myMap.put(4, "Summer");
myMap.put(5, "Morty");
```

The put method call requires two parameters:

- **The key:**
  - In this example it is of data type Integer
- **The value:**
  - In this example it is of data type String
- On the highlighted line, we inserted an entry with a key of 1 and a value of Rick.

Maps: containsKey method

The containsKey method returns a boolean indicating if the key exists.

```
Map <String, String> reservations = new HashMap<>();

reservations.put("A", "Rick");
reservations.put("B", "Beth");
reservations.put("C", "Jerry");

System.out.println(reservations.containsKey("A"));
// True
System.out.println(reservations.containsKey("G"));
// False
System.out.println(reservations.containsKey("Jerry"));
// False
```

- The containsKey method requires one parameter, the key you are searching for.
- containsKey returns a boolean

Note that the last example returns false because it's not a key, it's a value

Maps: get method

The get method returns the value associated with a key.

```
Map <String, String> reservations =
                                    new HashMap<String, String>();

reservations.put("A", "Rick");
reservations.put("B", "Beth");
reservations.put("C", "Jerry");

String name = reservations.get("A");
System.out.println(name); // Prints Rick

String anotherName = reservations.get(“K”);
System.out.println(anotherName); // Prints null
```

- The get method requires one parameter, the key you are searching for.
- It will return the value associated with the key.
- If the key is not present, a null is returned.

Maps: remove method

The remove method removes an item from the map, given a key

```
Map <String, String> reservations =
                                    new HashMap<String, String>();

reservations.put("A", "Rick");
reservations.put("B", "Beth");
reservations.put("C", "Jerry");

System.out.println(reservations.get("C"));
// Prints Jerry
reservations.remove("C");
System.out.println(reservations.get("C"));
 // Prints null
```

The remove method requires one parameter, the key you are searching for.

Maps: size method

The size method lists the size of the map.

```
Map <String, String> reservations =
                                        new HashMap<String, String>();

reservations.put("A", "Rick");
reservations.put("B", "Beth");
reservations.put("C", "Jerry");

System.out.println(reservations.size()); // Prints 3
reservations.remove("C");
System.out.println(reservations.size()); // Prints 2
```

- The size method requires no parameters.
- It will return an integer, the number of key-value pairs present.

Maps: Some Additional Rules

- Do not use primitive types with Maps, use the Wrapper classes instead.
- Make sure there are no duplicate keys!
  - If a key exists, a put against that key overwrites the old value with the new one.

Let’s work with some maps

Sets: Introduction

A set is also a collection of data.

- It differs from other collections we've seen so far in that no duplicate elements are allowed.
- It is also **unordered**.

Sets: Declaring

The following pattern is used in declaring a set.

```
import java.util.HashSet;
import java.util.Set;

public class MyClass {

        public static void main(String args[]) {

                Set<Integer> primeNumbersLessThan10 = new HashSet<Integer>();
        }
}
```

Note the we will need these 2 imports for a hash map.

We are creating a type of Set called a HashSet

We have specified that the set will contain only integers.

Note the “**new**” keyword which **instantiates** the set.

Sets: add method

The add method creates a new element in the set.

```
Set<Integer> primeNumbersLessThan10 =
                                        new HashSet<Integer>();
primeNumbersLessThan10.add(2);
primeNumbersLessThan10.add(3);
primeNumbersLessThan10.add(5);
```

Only one parameter is required, the data that is being added.

In this example I have specified that this is a set of Integers, so the integers 2, 3, and 5 are being added.

Sets: contains method

The contains method returns a boolean specifying if an element is part of the set.

```
Set<Integer> primeNumbersLessThan10 =
                                        new HashSet<Integer>();
primeNumbersLessThan10.add(2);
primeNumbersLessThan10.add(3);
primeNumbersLessThan10.add(5);

System.out.println(primeNumberLessThan10.contains(5));
// true
System.out.println(primeNumberLessThan10.contains(4));
// false
```

Only one parameter is required, the data that we want to search for.

Sets: remove method

The contains method returns a boolean specifying if an element is part of the set.

```
Set<Integer> primeNumbersLessThan10 =
                                        new HashSet<Integer>();
primeNumbersLessThan10.add(2);
primeNumbersLessThan10.add(3);
primeNumbersLessThan10.add(5);

primeNumbersLessThan10.remove(5);
```

Only one parameter is required, the data that we want to remove.

Sets: size method

Last but not least, sets also have a size method.

```
Set<Integer> primeNumbersLessThan10 =
                                        new HashSet<Integer>();
primeNumbersLessThan10.add(2);
primeNumbersLessThan10.add(3);
primeNumbersLessThan10.add(5);

System.out.println(primeNumbersLessThan10.size());
// 3
```

- No parameters are required.
- An integer is returned.

The keySet method for maps

```
Map <String, String> addresses = new HashMap<String, String>();

addresses.put("Drew", "100 Dearhaven Street");
addresses.put("Amy", "823 Flying Cow Lane");
addresses.put("Robert", "129 Decrepit Circle");
addresses.put("Robert", "499 Pothole Street");

Set <String> theKeys = addresses.keySet();

for (String keys : theKeys) {
        System.out.println(keys); // Drew Robert Amy (not necessarily in that order)
}
```

- Recall that maps contain keys (which must unique), and that a set has no repeating elements.
- We can use the keySet() method on a map to extract all the keys into a set.

Let’s work with some sets

Arrays vs Lists vs Maps vs Sets

- Use **Arrays** when you know the maximum number of elements, and you know you will primarily be **working with primitive data types**.
- Use **Lists** when you want something that works like an array, but you don't know the exact number of elements.
- Use **Maps** when you have key value pairs, where the keys are unique.
- Use **Sets** when you know your data does not contain repeating elements.
    - Coincidentally, all the keys on a given map comprise a set.