Module 1-8

Collections: Maps and Sets

Module 1 Day 8

Can you?

1. … explain the Map<T, T> data structure, its rules, and limitations
2. … perform the following tasks associated with Maps:
    a. Declare and initialize a Map
    b. Add and Retrieve values from the Map using the Keys
    c. Retrieve the Key set from a Map
    d. Check for Key uniqueness
    e. Iterate through the Key-Value-Pairs
    f. Remove items from the Map
3. … explain the conditions under which you would choose to use
    a. A Map vs. an Array or List
    b. A List vs. a Map or Array
    c. An Array vs. a List or Map
    (Data-types, Mutability, and Access Methods (index v Key) all come into play in the decision)

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 a type of **unordered** map called a HashMap.

Maps: Declaring

Map declarations follow this pattern.

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

public class MyClass {

        public static void main(String args[]) {

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

Note the 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, "Mortimer");
```

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("HY234-9234", "Rick");
reservations.put("HY234-4235", "Beth");
reservations.put("HY234-3234", "Jerry");

System.out.println(reservations.containsKey("HY234-4235"));
// True
System.out.println(reservations.containsKey("AAAI-4235"));
// 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 in 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 the key provided.

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

reservations.put("HY234-9234", "Rick");
reservations.put("HY234-4235", "Beth");
reservations.put("HY234-3234", "Jerry");

String name = reservations.get("HY234-9234");
System.out.println(name); // Prints Rick

String anotherName = reservations.get(“AAI93-2345”);
System.out.println(name); // Prints null
```

- The get method requires one parameter, the key you are searching for.
- It will return the value associated with the key.
- If keys match the parameter provided, it returns a null.

Maps: remove method

The remove method removes an item from the map using a key value.

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

reservations.put("HY234-9234", "Rick");
reservations.put("HY234-4235", "Beth");
reservations.put("HY234-3234", "Jerry");

System.out.println(reservations.get("HY234-3234"));
// Prints Jerry
reservations.remove("HY234-3234");
System.out.println(reservations.get("HY234-3234"));
 // Prints null
```

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

Maps: size method

The size method returns the number of key-value-pairs in the Map.

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

reservations.put("HY234-9234", "Rick");
reservations.put("HY234-4235", "Beth");
reservations.put("HY234-3234", "Jerry");

System.out.println(reservations.size()); // Prints 3
reservations.remove("HY234-3234");
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: The Rules

Maps are used to store key value pairs.

- Do not use primitive types with Maps, use the Wrapper classes instead.
- Make sure there are no duplicate keys. If a key value pair is entered with a key that already exists, it will overwrite the existing one!
    - As a corollary of the previous rule, you are allowed one null in your key set before your data is changed in an unexpected manner

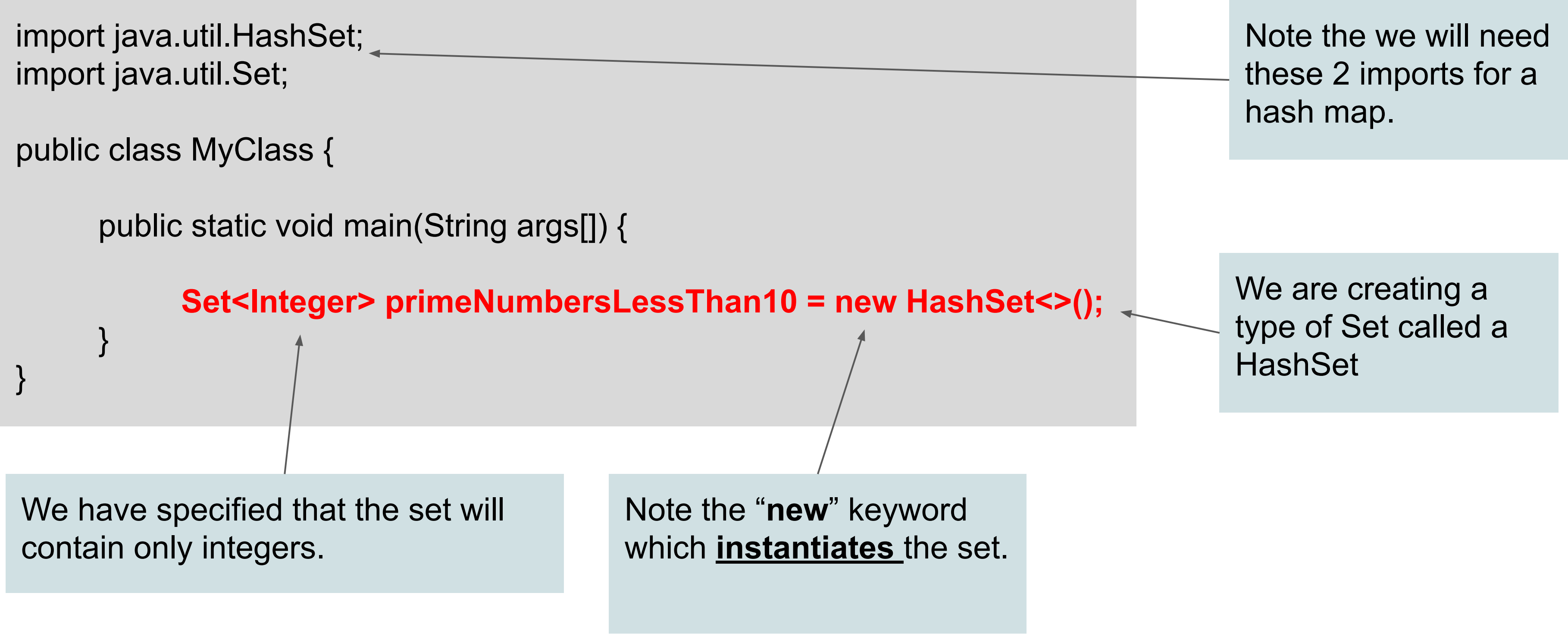
Sets: Introduction

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<>();
        }
}
```

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<>();
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<>();
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<>();
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<>();
primeNumbersLessThan10.add(2);
primeNumbersLessThan10.add(3);
primeNumbersLessThan10.add(5);

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

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

Making the Decision: 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 maximum number of elements.
- Use **Maps** when … you have key value pairs.
- Use **Sets** when … you know your data does not contain repeating elements.

** This "rule" is debatable in that you ***can*** declare Object[] arrays; they have their place but List<T> is far more common and meets the majority of use-cases.