Module 1-8

Collections II: 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. They are another form of in-memory data structures.

- Examples of key value pairs: dictionary entries (word -> definition), a phone book (name -> phone number), a list of employees (employee number -> employee name), a *vending machine slot and its items* (A3 -> Chips)

- Think of the keys as unique identifiers for a specific value

- We will focus on one 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.

1. Do not use primitive types with Maps, use Wrapper classes instead.
2. There cannot be duplicate keys. If a key value pair is entered with a key that already exists, it will overwrite the existing one!
3. Following Rule #2, you are also allowed just one null in your key set.
4. While null values may happen, avoid the use of null as a value.

```
Map<Integer, Integer> testMap = new HashMap<>();

testMap.put(1, null);
System.out.println( "Key 1 contains: " + testMap.get(1)); // Prints: Key 1 contains: null
System.out.println( "Key 6 contains: " + testMap.get(6)); // Prints: Key 6 contains: null

// We know that key 6 does not exist, so which null is "real"/intentional?
// For that matter, absent knowledge of the keys, which key is real?
```

Sets: Introduction

A set is another type of collection.

Sets differ from other collections we've seen so far in that they do not allow duplicate elements.

- Sets are also **unordered**.
- Sets do not allow duplicate data
- The Keys of a Map Collection are a Set, the map's keyset.

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, you care about or need the index, 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, need to add and remove elements, or you don't care about or need the index.
- 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.