Collections
Part 2: Maps & Sets

Today's Objectives

- Map<T, T>: rules and limitations
- Map tasks:
  - Declaring and initialize a Map
  - Adding and Retrieving values from the Map using the Keys
  - Retrieving the Key set from a Map
  - Checking for Key uniqueness
  - Iterating through the Key-Value-Pairs
  - Removing items from the Map
- When to use
  - A Map vs. an Array or List
  - A List vs. a Map or Array
  - An Array vs. a List or Map
    (Data-types, Mutability, and Access Methods (index vs key) all come into play in the decision)

Introducing: Maps

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)
- Think of the keys as unique identifiers for a specific value
- We will focus on a type of unordered map called a HashMap.

Declaring a Map

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<>();
     }
}
```

Declaring a Map

Map declarations follow this pattern:

`Map` and `HashMap` need to be imported from the `java.util` package.

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

public class MyClass {

     public static void main(String args[]) {

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

Declaring a Map

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<>();
     }
}
```

`Map` and `HashMap` need to be imported from the `java.util` package.

This indicates the key will be an `Integer` and the value will be a `String`.

Declaring a Map

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<>();
     }
}
```

`Map` and `HashMap` need to be imported from the `java.util` package.

This indicates the key will be an `Integer` and the value will be a `String`.

Remember the `new` keyword creates an **instance** of a class.

Declaring a Map

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<>();
     }
}
```

Map and **HashMap** need to be imported from the **java.util** package.

We are creating a **HashMap** implementation of **Map**.

This indicates the key will be an **Integer** and the value will be a **String**.

Remember the **new** keyword creates an **instance** of a class.

Adding an Entry to a Map

The **`put`** method adds a key/value pair entry 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(4, "Winter");
```

Adding an Entry to a Map

The **`put`** method adds a key/value pair entry 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(4, "Winter");
```

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`**.

Adding an Entry to a Map

The **`put`** method adds a key/value pair entry 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(4, "Winter");
```

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`**.

Here, we inserted an entry with a key of 1 and a value of Rick.

Adding an Entry to a Map

The **`put`** method adds a key/value pair entry 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(4, "Winter");
```

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`**.

If we insert a **duplicate key**, the key's **value will be overwritten**!!! So in this case **key 4** would return **Winter**.

Here, we inserted an entry with a key of 1 and a value of Rick.

Retrieving a Value by Its Key

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
```

Retrieving a Value by Its Key

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.

Retrieving a Value by Its Key

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.

Retrieving a Value by Its Key

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 no keys match the parameter provided, it returns a **`null`**.

Checking if an Key Exists in a Map

The **`containsKey`** method returns a **`boolean`** indicating if the key exists 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.containsKey("HY234-4235"))
;
// True
System.out.println(reservations.containsKey("AAAI-4235"));
// False
System.out.println(reservations.containsKey("Jerry"));
// False
```

Checking if an Key Exists in a Map

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

Checking if an Key Exists in a Map

The **`containsKey`** method returns a **`boolean`** indicating if the key exists 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.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 this example. **`false`** is returned because "Jerry" is not one of the keys in the **`Map`** (although it **IS** a value).

Finding the Number of Elements in a Map

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
```

Finding the Number of Elements in a Map

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 in the **`Map`**.

Getting a Map's Keys

We can get all the **`Map`**'s keys by using the **`keySet()`** method..

```
Set<String> keys = myMap.keySet();

for(String key : keys) {
    System.out.println("key: " + key);
}
```

Getting a Map's Keys

We can get all the **`Map`**'s keys by using the **`keySet()`** method..

```
Set<String> keys = myMap.keySet();

for(String key : keys) {
    System.out.println("key: " + key);
}
```

The **`keySet()`** method returns a collection of the type specified as the key in **`Map<T, T>`**

Getting a Map's Keys

We can get all the **`Map`**'s keys by using the **`keySet()`** method..

```
Set<String> keys = myMap.keySet();

for(String key : keys) {
    System.out.println("key: " + key);
}
```

The **`keySet()`** method returns a collection of the type specified as the key in **`Map<T, T>`**

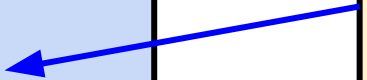
We can iterate through the key data using a for-each loop.

Iterating Using A Map's EntrySets

We can iterate through a **`Map`**'s Entries by using its **`EntrySet`**.

```
Map<Integer, String> myMap = new HashMap<Integer, String>();
myMap.put(1, "Hi");
myMap.put(100, "100%");
myMap.put(22, "Yay!");

for(Map.Entry<Integer, String> entry : myMap.entrySet() ) {
     Integer key = entry.getKey();
     String value = entry.getValue();
     System.out.println("key: " + key + " value: " + value);
}
```



The **`Map`**'s **`entrySet()`** method returns the **`EntrySet`**.

Iterating Using A Map's EntrySets

We can iterate through a **`Map`**'s Entries by using its **`EntrySet`**.

```
Map<Integer, String> myMap = new HashMap<Integer, String>();
myMap.put(1, "Hi");
myMap.put(100, "100%");
myMap.put(22, "Yay!");

for(Map.Entry<Integer, String> entry : myMap.entrySet() ) {
     Integer key = entry.getKey();
     String value = entry.getValue();
     System.out.println("key: " + key + " value: " + value);
}
```

The **`Map`**'s **`entrySet()`** method returns the **`EntrySet`**.

We can iterate through the **`EntrySet`** using a for-each loop with each entry returning an object of type **`Map.Entry<T, T>`**

Iterating Using A Map's EntrySets

We can iterate through a **`Map`**'s Entries by using its **`EntrySet`**.

```
Map<Integer, String> myMap = new HashMap<Integer, String>();
myMap.put(1, "Hi");
myMap.put(100, "100%");
myMap.put(22, "Yay!");

for(Map.Entry<Integer, String> entry : myMap.entrySet() ) {
     Integer key = entry.getKey();
     String value = entry.getValue();
     System.out.println("key: " + key + " value: " + value);
}
```

The **`Map`**'s **`entrySet()`** method returns the **`EntrySet`**.

We can get the key and value from the **`Entry`** using its **`getKey()`** and **`getValue()`** methods

We can iterate through the **`EntrySet`** using a for-each loop with each entry returning an object of type **`Map.Entry<T, T>`**

Removing an Entry from a Map

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
```

Removing an Entry from a Map

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.

Removing an Entry from a Map

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.

Note calling the **`get`** method with the key that was removed will now return **`null`** because the key is no longer in the **`Map`**.

Rules for Using Maps

`Maps` are used to store key value pairs.

- Do not use primitive types with **`Map`**s, 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 this rule, you are allowed one **`null`** in your key set before your data is changed in an unexpected manner

Introducing: Sets

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.

Declaring a Set

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<>();
     }
}
```

Declaring a Set

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<>();
     }
}
```

`Set` and `HashSet` need to be imported from the `java.util` package.

Declaring a Set

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<>();
     }
}
```

`Set` and `HashSet` need to be imported from the `java.util` package.

This indicates the `Set` will contain `Integers`.

Declaring a Set

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<>();
     }
}
```

`Set` and `HashSet` need to be imported from the `java.util` package.

This indicates the `Set` will contain `Integer`s.

Remember the `new` keyword creates an **instance** of a class.

Declaring a Set

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<>();
     }
}
```

`Set` and **`HashSet`** need to be imported from the **`java.util`** package.

We are creating a **`HashSet`** implementation of **`Set`**.

This indicates the **`Set`** will contain **`Integer`**s.

Remember the **`new`** keyword creates an **instance** of a class.

Adding an Element to a Set

The **`add`** method adds an element to the **`Set`**. The data type must match the declaration.

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

Adding an Element to a Set

The **`add`** method adds an element to the **`Set`**. The data type must match the declaration.

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

- **`add`** only requires one parameter: the data that is being added.

Adding an Element to a Set

The **`add`** method adds an element to the **`Set`**. The data type must match the declaration.

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

- **`add`** only requires one parameter: the data that is being added.

This example specifies that this is a **`Set`** of **`Integer`**s, so the integers 2, 3, and 5 are being added.

Checking If an Element is Contained in a Set

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
```

Checking If an Element is Contained in a Set

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
```

- **`contains`** only requires one parameter: the data that we want to search for.

Checking If an Element is Contained in a Set

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
```

- **`contains`** only requires one parameter: the data that we want to search for.

If the data specified by **`contains`** is found in the **`Set`**, **`true`** will be returned. Otherwise **`false`** will be returned.

Removing an Element from a Set

The **`remove`** method removes an element from a **`Set`**.

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

primeNumbersLessThan10.remove(5);
```

Removing an Element from a Set

The **`remove`** method removes an element from a **`Set`**.

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

primeNumbersLessThan10.remove(5);
```

- **`remove`** only requires one parameter: the data that is being removed.

Removing an Element from a Set

The **`remove`** method removes an element from a **`Set`**.

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

primeNumbersLessThan10.remove(5);
```

- **`remove`** only requires one parameter: the data that is being removed.

This will remove the element that is the **`Integer`** 5.;

Checking the Number of Elements in a Set

Last but not least, **`Set`**s 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
```

Checking the Number of Elements in a Set

Last but not least, **`Set`**s 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
```

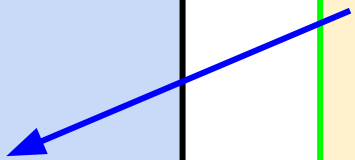
- No parameters are required.

Checking the Number of Elements in a Set

Last but not least, **`Set`**s 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

Checking the Number of Elements in a Set

Last but not least, **`Set`**s 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.

This will the number of elements, which is 3.

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.

Exercise Notes

- You should be checking for **`null`** values.
- You will need to create the Map data in the method you are writing for some of the problems.
- You may need to change some data and use some String methods to manipulate data in order to meet some of the requirements.
- Some problems are stated as being in cents, meaning when a value of $1 is given, the value will be represented as 100 cents.