

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");
```

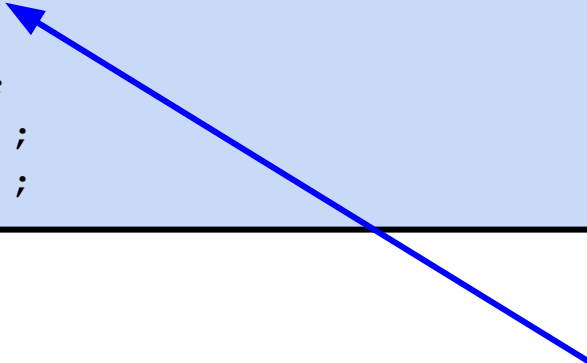
The **put** method call requires two parameters:

- 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:

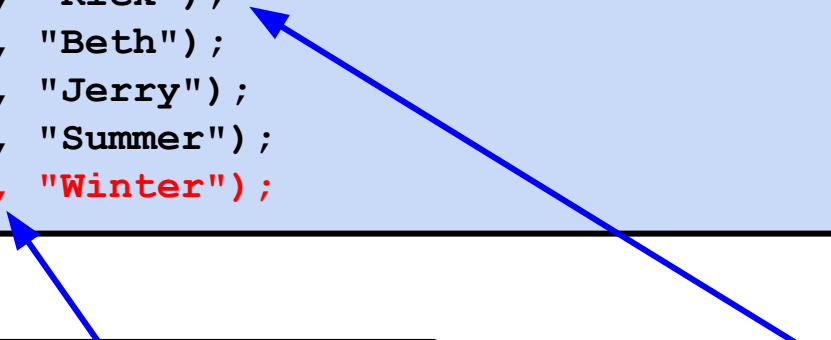
- The key: in this example it is of data type **Integer**.
- The value: in this example it is of data type **String**.

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

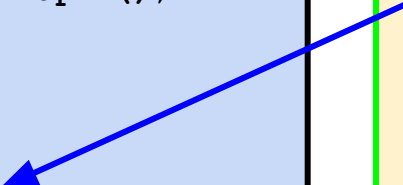
The **get** method returns the value associated with the key provided.

```
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.

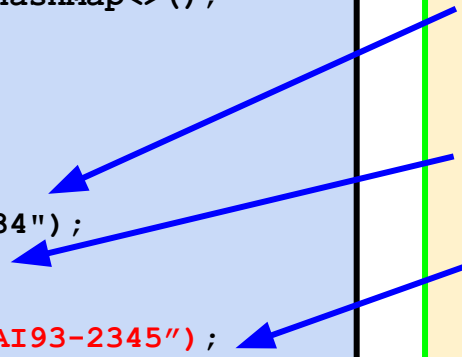
```
Map <String, String> reservations = new HashMap<>();  
  
reservations.put("HY234-9234", "Rick");  
reservations.put("HY234-4235", "Beth");  
reservations.put("HY234-3234", "Jerry");  
  
String name = reservations.get("HY234-9234");  
System.out.println(name); // Prints Rick  
  
String anotherName = 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/REMOVING ELEMENTS IN A MAP

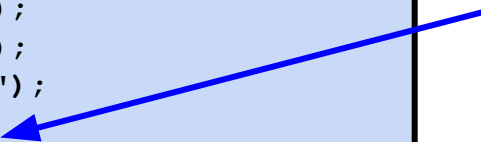
The **size** method returns the number of key-value-pairs in the **Map**.

```
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
```

FINDING THE NUMBER OF/REMOVING 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
```



- 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>**

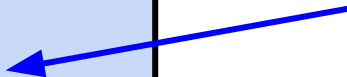
We can iterate through the key data using a for-each loop.

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
```

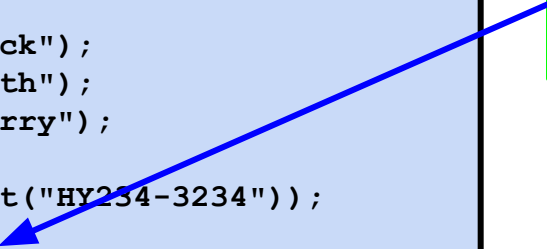
REMOVING AN ENTRY FROM A MAP

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 **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 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 a **List** 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 `Integers`.

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.

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

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

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

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.

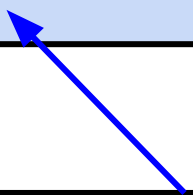
```
Set<Integer> primeNumbersLessThan10 = new HashSet<>();  
primeNumbersLessThan10.add(2);  
primeNumbersLessThan10.add(3);  
primeNumbersLessThan10.add(5);
```

- **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);
```



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

- **add** only requires one parameter: the data that is 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(primeNumbersLessThan10.contains(5));  
// true  
System.out.println(primeNumbersLessThan10.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(primeNumbersLessThan10.contains(5));  
// true  
System.out.println(primeNumbersLessThan10.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(primeNumbersLessThan10.contains(5));  
// true  
System.out.println(primeNumbersLessThan10.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);
```

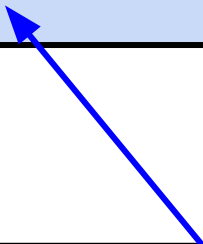
- **remove** only requires one parameter: the data that is being removed.

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

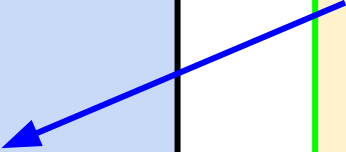
Last but not least, **Sets** also have a **size** method.

```
Set<Integer> primeNumbersLessThan10 = new HashSet<>();  
primeNumbersLessThan10.add(2);  
primeNumbersLessThan10.add(3);  
primeNumbersLessThan10.add(5);  
  
System.out.println(primeNumbersLessThan10.size());  
// 3
```

CHECKING THE NUMBER OF ELEMENTS IN A SET

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.
- 

CHECKING THE NUMBER OF ELEMENTS IN A SET

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

CHECKING THE NUMBER OF ELEMENTS IN A SET

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.

This will the print 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.
- 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.

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.