In the previous lesson, we looked at the `Optional<T>` class. You learned what an `Optional` is and how to create it.

In this lesson, we will look at all the operations that we can perform using an `Optional`.

Below is the list of methods available in the `Optional` class.

1) isPresent()

The `isPresent()` method is used to check if the optional contains a value or if it is null.

The method `isPresent()` returns the value true in case he id of the `Optional` objects contains a non-null value. Otherwise, it returns a false value.

```
Optional<Person> optional = getPerson();
if(optional.isPresent()){
    System.out.println(optional.get.getName())
}
```

2) ifPresent(Consumer<? super T> consumer)

Here is the syntax of `ifPresent()` method

```
public void ifPresent(Consumer<? super T> consumer)
```

It takes in a `Consumer` as a parameter and returns nothing. When `ifPresent()` is called, if a value is present, the specified consumer is invoked with the value. Otherwise, nothing happens.

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

public class StreamDemo {

    Map<Integer, Employee> empMap = new HashMap<>();

    public void populateEmployee() {
        empMap.put(123, new Employee("Alex", 23, 12000));
    }

    public Optional<Employee> getEmployee(Integer employeeId) {
        // Before returning the employee object we are wrapping it into an Optional
        return Optional.ofNullable(empMap.get(employeeId));
    }

    public static void main(String[] args) {
        StreamDemo demo = new StreamDemo();
        demo.populateEmployee();
        Optional<Employee> emp = demo.getEmployee(123);
        emp.ifPresent(System.out::println);
    }
}

class Employee {
    String name;
    int age;
```

Run Save Reset

3) get()

The `get()` method returns a value if it is present in this `Optional`. Otherwise, it throws NoSuchElementException.

It is risky to use this method without checking if the value is present or not using `isPresent()` method.

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

public class OptionalDemo {

    public static void main(String[] args) {

        Optional<String> optional = Optional.ofNullable(null);
        // this will throw exception because optional contains a null value.
        System.out.println(optional.get());
    }
}
```

Run Save Reset

4) orElse(T other)

This method returns the value present in the optional. If no value is present, then a default value provided as a parameter is returned.

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

public class OptionalDemo {

    public static void main(String[] args) {

        Optional<String> optional = Optional.ofNullable(null);
        // this will return the default value.
        System.out.println(optional.orElse("default string"));
    }
}
```

Run Save Reset

5) orElseGet(Supplier<? extends T> other)

This method returns the value present in the optional. If no value is present, then the value calculated from the supplier provided as a parameter is returned.

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

public class OptionalDemo {

    public static String getDefaultValue(){
        return "default";
    }

    public static void main(String[] args) {

        Optional<String> optional = Optional.ofNullable(null);
        // this will return the default value.
        System.out.println(optional.orElseGet(OptionalDemo::getDefaultValue));
    }
}
```

Run Save Reset

6) orElseThrow(Supplier<? extends T> other)

This method returns the value present in the optional. If no value is present, then it throws the exception created by the provided supplier.

```
import java.util.Optional;

public class OptionalDemo {

    public static void main(String[] args) {

        Optional<String> optional = Optional.ofNullable(null);
        // This will throw exception
        try {
            System.out.println(optional.orElseThrow(() -> new Exception("Resource not found.")));
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

Run Save Reset

7) Optional<T> filter(Predicate<? super T> predicate)

The `filter()` method is used to check if the value in our optional matches a particular condition. If yes, then the optional with the value is returned. Otherwise, an empty optional is returned.

```
import java.util.Optional;

public class OptionalDemo {

    public static void main(String[] args) {

        Optional<String> optional = Optional.ofNullable("orange");
        // since the filter condition is matched, this will return the optional.
        System.out.println(optional.filter(str -> str.equals("orange")));

        // Since the filter condition is not matched, this will return empty optional.
        System.out.println(optional.filter(str -> str.equals("apple")));

    }
}
```

Run Save Reset

8) map(Function<? super T, ? extends U> mapper)

As per Java docs, *"If a value is present, apply the provided mapping function to it, and if the result is non-null, return an `Optional` describing the result. Otherwise, return an empty `Optional`."*

```
import java.util.*;

public class StreamDemo {

    public static void main(String[] args) {

        // Creating an Optional of Employee object.
        Optional<Employee> optional = Optional.of(new Employee("Adam", 54, 20000));

        Optional
                .map(emp -> emp.getSalary()) // Fetching the salary from employee object.
                .filter(sal -> sal > 10000) // checking if the salary is greater than 10000.
                .ifPresent(System.out::println);
    }
}

class Employee {
    String name;
    int age;
    int salary;

    Employee(String name) {
        this.name = name;
    }

    Employee(String name, int age, int salary) {
        this.name = name;
        this.age = age;
```

Run Save Reset

9) flatMap(Function<? super T, Optional<U>> mapper)

Similar to the `map()` method, we also have the `flatMap()` method as an alternative for transforming values.

The difference is that the map transforms values only when they are unwrapped, whereas flatMap takes a wrapped value and unwraps it before transforming it.

Let's take the same example that we discussed while looking at `map()`. There is a slight modification though. The `getSalary()` method will return `Optional<Address>`, so the return type of **optional.map(emp -> emp.getSalary())** operation will be `Optional<Optional<Integer>>` .

```
Optional<Optional<Integer>> op1 = optional.map(emp -> emp.getSalary());
```

If we don't need a nested `Optional`, then we can use a `flatMap()`.

```
Optional<Integer> op1 = optional.flatMap(emp -> emp.getSalary());
```

Here is the complete code example.

```
import java.util.*;

public class OptionalDemo {

    public static void main(String[] args) {

        // Creating an Optional of Employee object.
        Optional<Employee> optional = Optional.of(new Employee("Adam", 54, 20000));
        optional.flatMap(emp -> emp.getSalary())
                .filter(sal -> sal > 10000)
                .ifPresent(System.out::println);
    }
}

class Employee {
    String name;
    int age;
    int salary;

    Employee(String name) {
        this.name = name;
    }

    Employee(String name, int age, int salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }
```

Run Save Reset