

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.

Method Summary	
Modifier and Type	Method and Description
static <T> Optional<T>	<code>empty()</code> Returns an empty Optional instance.
boolean	<code>equals(Object obj)</code> Indicates whether some other object is "equal to" this Optional.
Optional<T>	<code>filter(Predicate&lt;? super T&gt; predicate)</code> If a value is present, and the value matches the given predicate, return an Optional describing the value, otherwise return an empty Optional.
<U> Optional<U>	<code>flatMap(Function&lt;? super T, Optional&lt;U&gt;&gt; mapper)</code> If a value is present, apply the provided Optional-bearing mapping function to it, return that result, otherwise return an empty Optional.
T	<code>get()</code> If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.
int	<code>hashCode()</code> Returns the hash code value of the present value, if any, or 0 (zero) if no value is present.
void	<code>ifPresent(Consumer&lt;? super T&gt; consumer)</code> If a value is present, invoke the specified consumer with the value, otherwise do nothing.
boolean	<code>isPresent()</code> Return true if there is a value present, otherwise false.
<U> Optional<U>	<code>map(Function&lt;? super T, ? extends U&gt; mapper)</code> 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.
static <T> Optional<T>	<code>of(T value)</code> Returns an Optional with the specified present non-null value.
static <T> Optional<T>	<code>ofNullable(T value)</code> Returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional.
T	<code>orElse(T other)</code> Return the value if present, otherwise return other.
T	<code>orElseGet(Supplier&lt;? extends T&gt; other)</code> Return the value if present, otherwise invoke other and return the result of that invocation.
<X extends Throwable> T	<code>orElseThrow(Supplier&lt;? extends X&gt; exceptionSupplier)</code> Return the contained value, if present, otherwise throw an exception to be created by the provided supplier.
String	<code>toString()</code> Returns a non-empty string representation of this Optional suitable for debugging.

## 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 the id of the `Optional` objects contains a non-null value. Otherwise, it returns a false value.

## 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.

```
1 import java.util.HashMap;
2 import java.util.Map;
3 import java.util.Optional;
4
5 public class StreamDemo {
6
7     Map<Integer, Employee> empMap = new HashMap<>();
8
9     public void populateEmployee() {
10         empMap.put(123, new Employee("Alex", 23, 12000));
11     }
12
13     public Optional<Employee> getEmployee(Integer employeeId) {
14         // Before returning the employee object we are wrapping it into an Optional
15         return Optional.ofNullable(empMap.get(employeeId));
16     }
17
18     public static void main(String[] args) {
19         StreamDemo demo = new StreamDemo();
20         demo.populateEmployee();
21         Optional<Employee> emp = demo.getEmployee(123);
22         emp.ifPresent(System.out::println);
23     }
24 }
25
26 class Employee {
27     String name;
28     int age;
```

## 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.

```
1 import java.util.HashMap;
2 import java.util.Map;
3 import java.util.Optional;
4
5 public class OptionalDemo {
6
7     public static void main(String[] args) {
8
9         Optional<String> optional = Optional.ofNullable(null);
10         // This will throw exception because optional contains a null value.
11         System.out.println(optional.get());
12     }
13 }
```

## 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.

```
1 import java.util.HashMap;
2 import java.util.Map;
3 import java.util.Optional;
4
5 public class OptionalDemo {
6
7     public static void main(String[] args) {
8
9         Optional<String> optional = Optional.ofNullable(null);
10         // This will return the default value.
11         System.out.println(optional.orElse("default string"));
12     }
13 }
```

## 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.

```
1 import java.util.HashMap;
2 import java.util.Map;
3 import java.util.Optional;
4
5 public class OptionalDemo {
6
7     public static String getDefaultValue(){
8         return "default";
9     }
10
11     public static void main(String[] args) {
12
13         Optional<String> optional = Optional.ofNullable(null);
14         // This will return the default value.
15         System.out.println(optional.orElseGet(OptionalDemo::getDefaultValue));
16     }
17 }
```

## 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.

```
1 import java.util.Optional;
2
3 public class OptionalDemo {
4
5     public static void main(String[] args) {
6
7         Optional<String> optional = Optional.ofNullable(null);
8         // This will throw exception
9         try {
10             System.out.println(optional.orElseThrow(() -> new Exception("Resource not found.")));
11         } catch (Exception e) {
12             e.printStackTrace();
13         }
14     }
15 }
```

## 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.

```
1 import java.util.Optional;
2
3 public class OptionalDemo {
4
5     public static void main(String[] args) {
6
7         Optional<String> optional = Optional.ofNullable("orange");
8         // Since the filter condition is matched, this will return the optional.
9         System.out.println(optional.filter(str -> str.equals("orange")));
10
11         // Since the filter condition is not matched, this will return empty optional.
12         System.out.println(optional.filter(str -> str.equals("apple")));
13     }
14 }
```

## 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`."

```
1 import java.util.*;
2
3 public class StreamDemo {
4
5     public static void main(String[] args) {
6
7         // Creating an Optional of Employee object.
8         Optional<Employee> optional = Optional.of(new Employee("Adam", 54, 20000));
9
10         optional
11             .map(emp -> emp.getSalary()) // Fetching the salary from employee object.
12             .filter(sal -> sal > 10000) // Checking if the salary is greater than 10000.
13             .ifPresent(System.out::println);
14     }
15 }
```

## 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>>`.

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

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

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

Here is the complete code example.

```
1 import java.util.*;
2
3 public class OptionalDemo {
4
5     public static void main(String[] args) {
6
7         // Creating an Optional of Employee object.
8         Optional<Employee> optional = Optional.of(new Employee("Adam", 54, 20000));
9
10         optional.flatMap(emp -> emp.getSalary())
11             .filter(sal -> sal > 10000)
12             .ifPresent(System.out::println);
13     }
14 }
```

```
15 class Employee {
16     String name;
17     int age;
18     int salary;
19
20     Employee(String name) {
21         this.name = name;
22     }
23
24     Employee(String name, int age, int salary) {
25         this.name = name;
26         this.age = age;
27         this.salary = salary;
28     }
29 }
```