Week 13 - Java Optional

Louis Botha, `louis.botha@tuni.fi`

In Java, `Optional` is a container object that may or may not contain a non-null value. It's a class that was introduced in Java 8 to provide a more expressive way of handling null values.
The purpose of `Optional` is to provide a way to avoid null pointer exceptions by explicitly indicating that a value may be absent. An `Optional` instance can either contain a non-null value, or it can be empty. It's essentially a wrapper around a value that can be null, and provides methods to handle both cases.

An Optional object can have one of the following possible states:

- **Present**: The Optional object does not represent absence. A value is in the Optional object and it can be accessed by invoking get().
- **Absent**: The Optional object does represent absence of a value; you cannot access its content with get().

For example, consider a method that returns a `String` value. If the method encounters an error and cannot return a valid value, it might return `null`. However, the caller of the method may not be aware of this and may attempt to use the returned value, resulting in a `NullPointerException`. By returning an `Optional<String>` instead, the method can explicitly indicate that it may not return a valid value, and the caller can use `Optional` methods to handle this case safely.

Here's an example of how to use `Optional` in Java:

```
Optional<String> optionalName = Optional.ofNullable(getName
());


// Check if a value is present
if (optionalName.isPresent()) {
    String name = optionalName.get();
    System.out.println("Name is: " + name);
} else {
```

```
    System.out.println("Name is not available");
}

// Handle the case where a value is absent
String defaultName = optionalName.orElse("John Doe");
System.out.println("Default name is: " + defaultName);
```

In this example, `optionalName` is an `Optional` object that contains a `String` value returned by the `getName()` method. We use the `isPresent()` method to check if a value is present, and the `get()` method to retrieve the value. If the value is absent, we use the `orElse()` method to provide a default value.

Creating an `Optional`

In Java, you can create an `Optional` object in several ways:

1. Using the `of` method:

You can use the `of` method to create an `Optional` object with a non-null value. For example:

```
Optional<String> optionalName = Optional.of("Alice");
```

In this example, `optionalName` is an `Optional` object that contains a `String` value of "Alice".

Note that if you pass `null` as the argument to `of` method, it will throw a `NullPointerException`.

2. Using the `ofNullable` method:

You can use the `ofNullable` method to create an `Optional` object with a value that may be null. For example:

```
String name = getName();
Optional<String> optionalName = Optional.ofNullable(name);
```

In this example, `getName()` method might return `null`, so we use `ofNullable` to create an `Optional` object that may or may not contain a `String` value.

3. Using the `empty` method:

You can use the `empty` method to create an empty `Optional` object that contains no value. For example:

```
Optional<String> emptyOptional = Optional.empty();
```

In this example, `emptyOptional` is an empty `Optional` object that contains no value. Once you have created an `Optional` object, you can use its methods to retrieve or manipulate the contained value safely.

Checking the presence of an `Optional` value

You can check the presence of an `Optional` value using the `isPresent()` method. This method returns `true` if the `Optional` object contains a non-null value, and `false` if it's empty.

Here's an example:

```
Optional<String> optionalName = Optional.ofNullable("Alice");

if (optionalName.isPresent()) {
    String name = optionalName.get();
    System.out.println("Name is: " + name);
} else {
    System.out.println("Name is not available");
}
```

In this example, we create an `Optional` object `optionalName` that contains the value "Alice". We use the `isPresent()` method to check if the `Optional` object contains a value. If it does, we retrieve the value using the `get()` method and print it to the console. If it doesn't, we print a message indicating that the name is not available.

It's important to always check the presence of an `Optional` value before attempting to retrieve it with the `get()` method, as calling `get()` on an empty `Optional` object will result in a `NoSuchElementException`.

You can use the `isEmpty()` method to check if an `Optional` object is empty, i.e., contains no value. This method returns `true` if the `Optional` object is empty, and `false` if it contains a value.

Here's an example:

```
Optional<String> optionalName = Optional.empty();

if (optionalName.isEmpty()) {
    System.out.println("Name is not available");
} else {
    String name = optionalName.get();
    System.out.println("Name is: " + name);
}
```

In this example, we create an empty `Optional` object `optionalName`. We use the `isEmpty()` method to check if the `Optional` object is empty. Since it is, we print a message indicating that the name is not available. If the `Optional` object contained a value, we would retrieve it using the `get()` method and print it to the console.

Note that the `isEmpty()` method was introduced in Java 11. If you're using an earlier version of Java, you can use the `!isPresent()` method to achieve the same result.

Default value `orElse()`

You can use the `orElse()` method to provide a default value for an `Optional` object in case it's empty. This method returns the value contained in the `Optional` object if it's not empty, or the default value if it's empty.

Here's an example:

```
Optional<String> optionalName = Optional.empty();
String name = optionalName.orElse("Unknown");

System.out.println("Name is: " + name);
```

In this example, we create an empty `Optional` object `optionalName`. We use the `orElse()` method to provide a default value of "Unknown" in case the `Optional` object is empty. The `orElse()` method returns the default value since the `Optional` object is empty, and we assign it to the `name` variable. We then print the value of `name` to the console, which is "Unknown".

Here's another example where the `Optional` object contains a value:

```
Optional<String> optionalName = Optional.of("Alice");
String name = optionalName.orElse("Unknown");


System.out.println("Name is: " + name);
```

In this example, we create an `Optional` object `optionalName` that contains the value "Alice". We use the `orElse()` method to provide a default value of "Unknown" in case the `Optional` object is empty, which it isn't. The `orElse()` method returns the value "Alice", and we assign it to the `name` variable. We then print the value of `name` to the console, which is "Alice".
Note that the `orElse()` method is useful for providing default values or fallback values when working with `Optional` objects, and can help avoid `NullPointerExceptions`.

Default value `orElseGet()`

You can use the `orElseGet()` method to provide a default value for an `Optional` object in case it's empty, but unlike `orElse()`, the default value is supplied by a `Supplier` function that is only called if the `Optional` object is empty. This can be useful when the default value is expensive to compute or has side effects.

Here's an example:

```
Optional<String> optionalName = Optional.empty();
String name = optionalName.orElseGet(() -> {
    // Perform some expensive computation or side-effect here
    return "Unknown";
});
```

```
System.out.println("Name is: " + name);
```

In this example, we create an empty `Optional` object `optionalName`. We use the `orElseGet()` method to provide a default value of "Unknown" in case the `Optional` object is empty. The `orElseGet()` method takes a `Supplier` function that returns the default value, which in this case is a lambda expression that performs some expensive computation or side-effect. Since the `Optional` object is empty, the `orElseGet()` method calls the lambda expression and returns the default value, which we assign to the `name` variable. We then print the value of `name` to the console, which is "Unknown".

Here's another example where the `Optional` object contains a value:

```
Optional<String> optionalName = Optional.of("Alice");
String name = optionalName.orElseGet(() -> {
    // This lambda expression is not executed because the Opti
onal object is not empty
    return "Unknown";
});

System.out.println("Name is: " + name);
```

In this example, we create an `Optional` object `optionalName` that contains the value "Alice". We use the `orElseGet()` method to provide a default value of "Unknown" in case the `Optional` object is empty, which it isn't. The `orElseGet()` method doesn't call the lambda expression since the `Optional` object is not empty, and returns the value "Alice", which we assign to the `name` variable. We then print the value of `name` to the console, which is "Alice".
Note that the `orElseGet()` method is useful for providing default values that are expensive to compute or have side effects, and can help improve the performance of your code.

Exceptions With `orElseThrow()`

You can use the `orElseThrow()` method to provide a custom exception to be thrown in case an `Optional` object is empty. This method returns the value contained in the `Optional` object if it's not empty, or throws a `NoSuchElementException` with the specified exception message if it's empty.

Here's an example:

```
Optional<String> optionalName = Optional.empty();

String name = optionalName.orElseThrow(() -> new NoSuchElement
Exception("Name is not available"));

System.out.println("Name is: " + name);
```

In this example, we create an empty `Optional` object `optionalName`. We use the `orElseThrow()` method to throw a `NoSuchElementException` with the message "Name is not available" in case the `Optional` object is empty. The `orElseThrow()` method takes a `Supplier` function that returns the exception to be thrown, which in this case is a lambda expression that creates a new `NoSuchElementException`. Since the `Optional` object is empty, the `orElseThrow()` method calls the lambda expression and throws the exception. The code doesn't reach the `println` statement since the exception is thrown.

Here's another example where the `Optional` object contains a value:

```
Optional<String> optionalName = Optional.of("Alice");

String name = optionalName.orElseThrow(() -> new NoSuchElement
Exception("Name is not available"));

System.out.println("Name is: " + name);
```

In this example, we create an `Optional` object `optionalName` that contains the value "Alice". We use the `orElseThrow()` method to return the value "Alice" in case the `Optional` object is not empty. The `orElseThrow()` method doesn't call the lambda expression since the `Optional` object is not empty, and returns the value

"Alice", which we assign to the `name` variable. We then print the value of `name` to the console, which is "Alice".

Note that the `orElseThrow()` method is useful for throwing custom exceptions with meaningful error messages when an `Optional` object is empty, and can help improve the error handling in your code.

Step-by-Step Program Example

Here's is an example of how to use Optionals in your Java program:

1. Start by importing the `java.util.Optional` class at the beginning of your Java file.

```
import java.util.Optional;
```

2. Declare a variable of type `Optional` and initialize it with either a value or `null` using the `ofNullable` method.

```
String name = "Alice";
Optional<String> optionalName = Optional.ofNullable(name);
```

In this example, we declare a variable `name` and initialize it with the value "Alice". We then create an `Optional` object `optionalName` using the `ofNullable` method, which takes a value and returns an `Optional` object containing that value. If the value is `null`, the method returns an empty `Optional` object.

3. Check whether the `Optional` object is empty or not using the `isPresent` method.

```
if (optionalName.isPresent()) {
    System.out.println("Name is: " + optionalName.get());
} else {
    System.out.println("Name is not available");
}
```

In this example, we use the `isPresent` method to check whether the `Optional` object `optionalName` contains a value or not. If the `Optional` object is not empty, we use the `get` method to retrieve the value and print it to the console. If the `Optional` object is empty, we print a message indicating that the name is not available.

4. Provide a default value using the `orElse` method.

```
String defaultName = "Unknown";
String name = optionalName.orElse(defaultName);
System.out.println("Name is: " + name);
```

In this example, we use the `orElse` method to provide a default value of "Unknown" in case the `Optional` object is empty. The method returns the value contained in the `Optional` object if it's not empty, or the default value if it's empty. We assign the result to the `name` variable and print it to the console.

5. Use the `map` method to apply a transformation to the value in the `Optional` object.

```
String greeting = optionalName.map(n -> "Hello " + n).orElse
("Hello Stranger");
System.out.println(greeting);
```

In this example, we use the `map` method to apply a transformation to the value contained in the `Optional` object. The `map` method takes a `Function` that maps the value to a new value, and returns a new `Optional` object containing the transformed value. If the `Optional` object is empty, the method returns an empty `Optional` object. In this example, we apply a transformation that adds "Hello " before the name, and provide a default value of "Hello Stranger" in case the `Optional` object is empty. We assign the result to the `greeting` variable and print it to the console.

That's a basic example of how to use Optionals in your Java program. You can use them to handle null values, avoid `NullPointerExceptions`, and provide default values or transformations in a safe and concise way.

Read More