Theory: Optional

604 users solved this topic. Latest completion was about 18 hours ago.

17 minutes 0 / 3 problems solved

Skip this topic Start practicing

§1. A billion-dollar mistake

Like many programming languages, Java uses `null` to represent the absence of a value. Sometimes this approach leads to exceptions like **NPE** while non-null checks make code less readable. The British computer scientist Tony Hoare—the inventor of the `null` concept—even describes intoducing `null` as a **"billion-dollar mistake"** since it has led to innumerable errors, vulnerabilities, and system crashes. To avoid the issues associated with `null`, Java provides the `Optional` class that is a safer alternative for standard `null` references.

§2. Optional values

The `Optional<T>` class represents the presence or absence of a value of the specified type `T`. An object of this class can be either **empty** or **non-empty**.

Let's look at an example. In the following code, we create two Optional objects called `absent` and `present`. The first object represents an empty value (such as `null`), and the second one keeps a real string value.

```
Optional<String> absent = Optional.empty();
Optional<String> present = Optional.of("Hello");
```

The `isPresent` method checks whether an object is empty or not:

```
System.out.println(absent.isPresent()); // false
System.out.println(present.isPresent()); // true
```

Starting with Java 11, we can also invoke the opposite `isEmpty` method.

If you pass the `null` object to the `of` method, it will cause **NPE**.

§3. Optionals and nullable objects

In a situation when you don't know whether a variable is `null` or not, you should pass it to the `ofNullable` method instead of the `of` method. It creates an empty Optional if the passed value is `null`.

The word `nullable` means that a variable is potentially `null`.

In the following example, the `getRandomMessage` method may return `null` or some string message. Depending on what is returned, the result will be different.

```
String message = getRandomMessage(); // it may be null

Optional<String> optMessage = Optional.ofNullable(message);

System.out.println(optMessage.isPresent()); // true or false
```

If the `message` is not `null` (e.g. `"Hello"`) the code will print `true`. Otherwise, it will print `false` because the Optional object is empty.

In a sense, `Optional` is like a box that contains either some value or nothing. It wraps a value or `null` keeping the possibility to check it by using special methods.

Current topic:

Optional

Topic depends on:

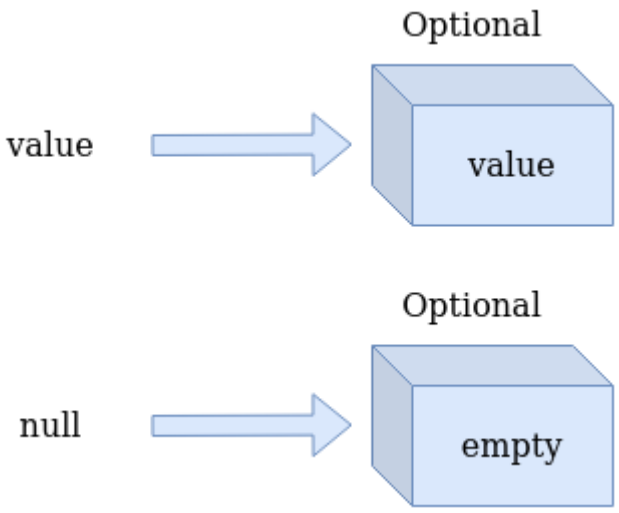
- ✓ NPE
- ✗ Generic programming
- ✗ Functional interfaces
- ✗ Standard functional interfaces

Topic is required for:

Functional data processing with streams

CRUD Repositories

Table of contents:



As you can see, the idea is pretty simple. Let's consider what we can do with Optionals.

It is important that assigning the `null` value to a variable of the `Optional` type is possible, but it is considered as a bad programming practice.

§4. Getting the value from an Optional

The most obvious thing to do with an Optional is to get its value. For now, we're going to discuss three methods with such purpose:

- `get` returns the value if it's present, otherwise throws an exception;
- `orElse` returns the value if it's present, otherwise returns `other`;
- `orElseGet` returns the value if it's present, otherwise invokes `other` and returns its result.

Let's see how they work. First, we use the `get` method to obtain the present value:

```
1 Optional<String> optName = Optional.of("John");
2 String name = optName.get(); // "John"
```

This code works well and returns the name `"John"` from the Optional. But if an Optional object is empty, the program throws `NoSuchElementException` exception.

```
1 Optional<String> optName = Optional.ofNullable(null);
2 String name = optName.get(); // throws NoSuchElementException
```

This is not exactly what we would expect from the class designed to reduce the number of exceptions.

Since Java 10, the preferred alternative to the `get` method is the `orElseThrow` method whose behavior is the same, but the name describes it better.

Other methods allow us to handle the empty Optional case. Let's consider them.

The `orElse` method is used to extract the value wrapped inside an Optional object or return some default value when the Optional is empty. The default value is passed to the method as its argument:

```
1 String nullableName = null;
2 String name = Optional.ofNullable(nullableName).orElse("unknown");
3
4 System.out.println(name); // unknown
```

Unlike the previous example, this one doesn't throw an exception, but returns a default value instead.

`orElseGet` method is quite similar, but it takes a **supplier function** to produce a result instead of taking some value to return:

```
String name = Optional
        .ofNullable(nullableName)
        .orElseGet(SomeClass::getDefaultResult);
```

In this example, we use the `getDefaultResult` method for producing a default result.

§5. Conditional actions

There are also convenient methods that take functions as arguments and perform some actions on values wrapped inside Optional:

- `ifPresent` performs the given action with the value, otherwise does nothing;
- `ifPresentOrElse` performs the given action with the value, otherwise performs the given empty-based action.

The `ifPresent` method allows us to run some code on the value if the Optional is not empty. The method takes a **consumer function** that can process the value.

The following example prints the length of a company's name by using the `ifPresent`.

```
Optional<String> companyName = Optional.of("Google");
companyName.ifPresent((name) -> System.out.println(name.length())); // 6
```

However, the following code does not print anything because the Optional object is empty.

```
Optional<String> noName = Optional.empty();
noName.ifPresent((name) -> System.out.println(name.length()));
```

It does not throw an exception due to performing the internal `null` check.

The "classic" equivalent of these two code snippets looks like the following:

```
String companyName = ...;
if (companyName != null) {
    System.out.println(companyName.length());
}
```

This code is more error-prone because it is possible to forget to perform the `null` check explicitly and then get the **NPE**.

The method `ifPresentOrElse` is a safer alternative to the whole `if-else` statement. It executes one of two functions depending on whether the value is present in the `Optional`.

```

Optional<String> optName = Optional.ofNullable(/* some value goes here */);

  optName.ifPresentOrElse(
          (name) -> System.out.println(name.length()),
          () -> System.out.println(0)
  );
```

If `optName` contains some value (like `"Google"`), the lambda expression is called and it prints the length of the name. If `optName` is empty, the second function prints `0` as the default value. Sometimes, developers call the second lambda expression **fallback** which is an alternative plan if something went wrong (no value).

§6. Conclusion

Objects of the `Optional` class represent the presence or absence of a value in a safer way than `null` does.

This class:

- allows programmers to avoid `null` references that may lead to **NPE**;

- reduces the boilerplate code for checking `null` (such as `if (something == null)`);
- provides a rich set of functional methods.

Now, you can choose the most appropriate way to represent a possibly missing value besides using `null`. In this topic, we haven't considered some advanced functional methods like `map`, `filter`, and `flatMap`, but they will be considered further.

Report a typo

65 users liked this theory. **2** didn't like it. **What about you?**

Start practicing

Comments (2) Hints (0) Useful links (0) Show discussion