# Optional in Java 8: Part 1

In this lesson, we will look at the newly introduced Optional class. We will also look at different ways of creating an Optional.

**We'll cover the following**



## What is an `Optional`? #

Java 8 has introduced a new class `Optional<T>` in the `java.util` package.

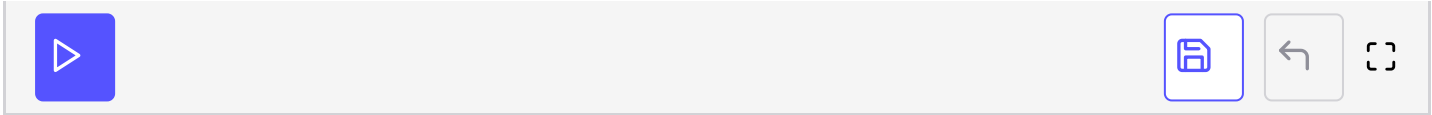
The `Optional<T>` is a wrapper class that stores an object of type `T`. The object may or may not be present in the optional.

According to Oracle, *"Java 8 `Optional` works as a container type for the value which is probably absent or null. Java `Optional` is a final class present in the `java.util` package."*

Let us look at how things worked before optional was introduced. In the below example, we have a `getEmployee()` method which gets the employee object from a `Map`. After fetching the employee object, we will print its details.

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

public class StreamDemo {

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

    public Employee getEmployee(Integer employeeId) {
        return empMap.get(employeeId);
    }

    public static void main(String[] args) {
        StreamDemo demo = new StreamDemo();
```

```
        //Fetching the employee with id 123. But since map is empty this will be null.
        Employee emp = demo.getEmployee(123);

        // This will throw Null Pointer Exception because emp is null
        System.out.println(emp.getName());
    }
}

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;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public int getSalary() {
        return salary;
    }

    @Override
    public String toString() {
        return "Employee{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", salary=" + salary +
                '}';
    }
}
```



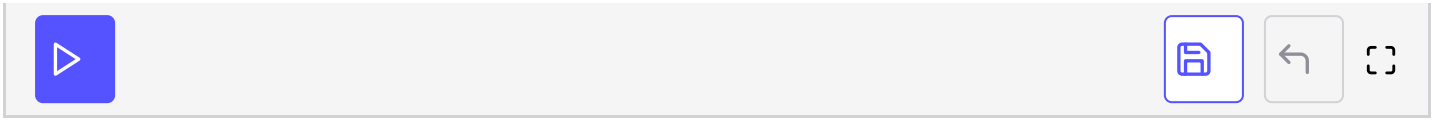
As you can see, every time we use an object there is a chance of that dreaded `NullPointerException`. To overcome this we need to add null checks, which result in a lot of boilerplate code. Using `Optional` makes the code more readable and less prone to error.

The below example shows how the same program can be written using an `Optional<T>`. At **line 11**, instead of directly returning the `Employee` object, we are

wrapping it into an `Optional`.

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

public class StreamDemo {

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

    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();
        Optional<Employee> emp = demo.getEmployee(123);
        // Before getting a value from Optional we check if the value is present through isPresent
        if(emp.isPresent()){
            System.out.println(emp.get().getName());  // We use get() method to get the value from
        } else{
            System.out.println("No employee returned.");
        }
    }
}

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;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public int getSalary() {
        return salary;
    }

    @Override
    public String toString() {
        return "Employee{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", salary=" + salary +
```

```
                '}';
    }
}
```



After looking at the above code, you might be wondering what the use of `Optional<T>` is if we need to check whether the value in the optional is null or not, using the `isPresent()` method. Why can't we just use the method directly and do a null check instead of wrapping it into an `Optional<T>`?

The benefit of `Optional<T>` is not that we are saved from applying a null check. The benefit is that `Optional<T>` class provides us lots of utility methods that we can apply to our wrapped objects.

## Different ways of creating an Optional #

There are three different ways of creating an `Optional` object.

### 1) Using `empty()` method. #

We can create an empty optional using the `empty()` method. The optional created through `empty()` will contain a null value.

```
Optional < Person > person = Optional.empty();
```

### 2) Using `of()` method #

We can create an `Optional` object that has a non-null value using `of()` method. If we create an `Optional` using the `of()` method and the value is null, then it will throw a `Null Pointer Exception`.

To create an `Optional` using the `of()` method, when you are really sure that the value is not null, do the following.

```
Person person = new Person();
Optional<Person> optional = Optional.of(person);
```

### 3) Using `ofNullable()` method #

If while creating the `Optional`, you are not sure if the value is null or not null, then use the `ofNullable()` method. If a non-null value is passed in `Optional.ofNullable()`, then it will return the `Optional`, containing the specified

value. Otherwise, it will return an empty `Optional`.

```
Person person = new Person();
Optional<Person> optional = Optional.ofNullable(person);
```

---

This lesson provided a basic introduction to what an `Optional` is. In the next lesson, we will look at all the methods present in the `Optional` class.