How to Mock, Spy, and Fake Spring Beans

by Lubos Krnac MVB · Jan. 08, 16 · Java Zone · Tutorial

EDIT: As of Spring Boot 1.4.0, faking of Spring Beans is supported natively via annotation @MockBean. Read Spring Boot docs for more info.

About a year ago, I wrote a blog post on how to mock Spring Beans. The patterns described there were a little bit invasive to the production code. As one of the readers Colin correctly pointed out in a comment. Today I'm introducing a better way to spy/mock Spring Beans based on the `@Profile` annotation. This blog post is going to describe this technique. I used this approach with success at work and also in my side projects.

Note that widespread mocking in your application is often considered as design smell.

Introducing Production Code

First of all, we need code under test to demonstrate mocking. We will use these simple classes:

```
@Repository
public class AddressDao {
    public String readAddress(String userName) {
        return "3 Dark Corner";
    }
}

@Service
public class AddressService {
    private AddressDao addressDao;

    @Autowired
    public AddressService(AddressDao addressDao) {
        this.addressDao = addressDao;
    }

    public String getAddressForUser(String userName){
        return addressDao.readAddress(userName);
    }
}

@Service
```

```
public class UserService {
    private AddressService addressService;

    @Autowired
    public UserService(AddressService addressService) {
        this.addressService = addressService;
    }

    public String getUserDetails(String userName){
        String address = addressService.getAddressForUser(userName);
        return String.format("User %s, %s", userName, address);
    }
}
```

Of course this code doesn't make much sense, but it will be good to demonstrate how to mock Spring Beans. `AddressDao` just returns a string and thus simulates a read from some data source. It is autowired into `AddressService`. This bean is autowired into `UserService`, which is used to construct a string with the user name and address.

Notice that we are using **constructor injection** because field injection is considered a bad practice. If you want to enforce constructor injection for your application, Oliver Gierke (Spring ecosystem developer and Spring Data lead) recently created very nice project called Ninjector.

Configuration to scan all these beans is pretty standard Spring Boot's main class:

```
@SpringBootApplication
public class SimpleApplication {
    public static void main(String[] args) {
        SpringApplication.run(SimpleApplication.class, args);
    }
}
```

Mock Spring Beans (Without AOP)

Let's test the `AddressService` class where we mock `AddressDao`. We can create this mock via Spring's `@Profiles` and **@Primary** annotations this way:

```
@Profile("AddressService-test")
@Configuration
public class AddressDaoTestConfiguration {
    @Bean
    @Primary
    public AddressDao addressDao() {
        return Mockito.mock(AddressDao.class);
    }
}
```

This test configuration will be applied only when Spring profile's `AddressService-test` is active. When it's applied, it registers a Bean with the type `AddressDao`, which is a mock instance created by **Mockito**. The `@Primary` annotation tells Spring to use this instance instead of a real one when somebody autowires the `AddressDao` Bean.

Test class is using **JUnit** framework:

```
@ActiveProfiles("AddressService-test")
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(SimpleApplication.class)
public class AddressServiceITest {
```

```
    @Autowired
    private AddressService addressService;

    @Autowired
    private AddressDao addressDao;

    @Test
    public void testGetAddressForUser() {
        // GIVEN
        Mockito.when(addressDao.readAddress("john"))
            .thenReturn("5 Bright Corner");

        // WHEN
        String actualAddress = addressService.getAddressForUser("john");

        // THEN
        Assert.assertEquals("5 Bright Corner", actualAddress);
    }
}
```

We activate the profile `AddressService-test` to enable `AddressDao` mocking. The annotation `@RunWith` is needed for Spring integration tests and `@SpringApplicationConfiguration` defines which Spring configuration will be used to construct the context for testing. Before the test, we autowire an instance of `AddressService` under test and the `AddressDao` mock.

Subsequent testing methods should be clear if you are using Mockito. In the `GIVEN` phase, we record the desired behavior into a mock instance. In the `WHEN` phase, we execute testing code, and in the `THEN` phase, we verify if testing code returned the value we expect.

Spy on Spring Beans (Without AOP)

For a spying example, we will be spying on the `AddressService` instance:

```
@Profile("UserService-test")
@Configuration
public class AddressServiceTestConfiguration {
    @Bean
    @Primary
    public AddressService addressServiceSpy(AddressService addressService) {
        return Mockito.spy(addressService);
    }
}
```

This Spring configuration will be component-scanned only if the profile `UserService-test` is active. It defines primary Bean of type `AddressService`. `@Primary` tells Spring to use this instance in case two Beans of this type are present in the Spring context. During the construction of this Bean, we autowire an existing instance of `AddressService` from the Spring context and use Mockito's spying feature. The Bean we are registering is effectively delegating all the calls to the original instance, but Mockito spying allows us to verify interactions on a spied instance.

We will test the behavior of `UserService` this way:

```
@ActiveProfiles("UserService-test")
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(SimpleApplication.class)
public class UserServiceITest {
    @Autowired
```

```
    private UserService userService;

    @Autowired
    private AddressService addressService;

    @Test
    public void testGetUserDetails() {
        // GIVEN - Spring scanned by SimpleApplication class

        // WHEN
        String actualUserDetails = userService.getUserDetails("john");

        // THEN
        Assert.assertEquals("User john, 3 Dark Corner", actualUserDetails);
        Mockito.verify(addressService).getAddressForUser("john");
    }
}
```

For testing, we activate the **`UserService-test`** profile so our spying configuration will be applied. We autowire **`UserService`**, which is under test, and **`AddressService`**, which is being spied via Mockito.

We don't need to prepare any behavior for testing in the **`GIVEN`** phase. The **`WHEN`** phase is obviously executing code under test. In the **`THEN`** phase, we verify if the testing code returned the value we expect and also if the **`addressService`** call was executed with correct parameters.

Problems With Mockito and Spring AOP

Let's say that we now want to use the Spring AOP module to handle some cross-cutting concerns. For example, to log calls on our Spring Beans in this way, we use this code:

```
package net.lkrnac.blog.testing.mockbeanv2.aoptesting;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.context.annotation.Profile;
import org.springframework.stereotype.Component;

import lombok.extern.slf4j.Slf4j;

@Aspect
@Component
@Slf4j
@Profile("aop") //only for example purposes
public class AddressLogger {
    @Before("execution(* net.lkrnac.blog.testing.mockbeanv2.beans.*.*(..))")
    public void logAddressCall(JoinPoint jp){
        log.info("Executing method {}", jp.getSignature());
    }
}
```

This AOP Aspect is applied before the call on Spring Beans from the package **`net.lkrnac.blog.testing.mockbeanv2`**. It is using Lombok's annotation **`@Slf4j`** to log the signature of the called method. Notice that this bean is created only when the **`aop`** profile is defined. We are using this profile to separate AOP and non-AOP testing examples. In a real application you wouldn't want to use such a profile.

We also need to enable AspectJ for our application, therefore all the following examples will be using this Spring Boot main class:

```
@SpringBootApplication
@EnableAspectJAutoProxy
public class AopApplication {
    public static void main(String[] args) {
        SpringApplication.run(AopApplication.class, args);
    }
}
```

AOP constructs are enabled by `@EnableAspectJAutoProxy` .

Such AOP constructs may be problematic if we combine Mockito for mocking with Spring AOP. That's because both use CGLIB to proxy real instances, and when the Mockito proxy is wrapped into the Spring proxy, we can experience type mismatch problems. These can be mitigated by configuring a Bean's scope with `ScopedProxyMode.TARGET_CLASS` , but Mockito `verify()` calls will still fail with `NotAMockException` . These problems can be seen if we enable the `aop` profile for `UserServiceITest` .

Mock Spring Beans Proxied by Spring AOP

To overcome these problems, we will wrap a mock into this Spring bean:

```
package net.lkrnac.blog.testing.mockbeanv2.aoptesting;

import org.mockito.Mockito;
import org.springframework.context.annotation.Primary;
import org.springframework.context.annotation.Profile;
import org.springframework.stereotype.Repository;

import lombok.Getter;
import net.lkrnac.blog.testing.mockbeanv2.beans.AddressDao;

@Primary
@Repository
@Profile("AddressService-aop-mock-test")
public class AddressDaoMock extends AddressDao{
    @Getter
    private AddressDao mockDelegate = Mockito.mock(AddressDao.class);

    public String readAddress(String userName) {
        return mockDelegate.readAddress(userName);
    }
}
```

`@Primary` makes sure that this Bean will take precedence over the real `AddressDao` Bean during injection. To make sure it will be applied only for specific tests, we define the profile `AddressService-aop-mock-test` for this Bean. It inherits the `AddressDao` class so that it can act as a full replacement of that type.

In order to fake behavior, we define a mock instance of type `AddressDao` , which is exposed via getters defined by Lombok's `@Getter` annotation. We also implement the `readAddress()` method, which is expected to be called during tests. This method just delegates the call to the mock instance.

The test where this mock is used can look like this:

```
@ActiveProfiles({"AddressService-aop-mock-test", "aop"})
```

```
@ActiveProfiles({"AddressService-aop-mock-test", "aop"})
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(AopApplication.class)
public class AddressServiceAopMockITest {
    @Autowired
    private AddressService addressService;

    @Autowired
    private AddressDao addressDao;

    @Test
    public void testGetAddressForUser() {
        // GIVEN
        AddressDaoMock addressDaoMock = (AddressDaoMock) addressDao;
        Mockito.when(addressDaoMock.getMockDelegate().readAddress("john"))
            .thenReturn("5 Bright Corner");

        // WHEN
        String actualAddress = addressService.getAddressForUser("john");

        // THEN
        Assert.assertEquals("5 Bright Corner", actualAddress);
    }
}
```

In the test we define the **`AddressService-aop-mock-test`** profile to activate **`AddressDaoMock`**, the **`aop`** profile and the **`AddressLogger`** AOP aspect. For testing, we autowire the testing Bean **`addressService`** and its faked dependency, **`addressDao`**. As we know, **`addressDao`** will have a type of **`AddressDaoMock`** because this bean was marked as **`@Primary`**. Therefore, we can cast it and record its behavior into **`mockDelegate`**.

When we call the testing method, recorded behavior should be used because we expect the testing method to use the **`AddressDao`** dependency.

Spy on a Spring Bean Proxied by Spring AOP

A similar pattern can be used for spying on the real implementation. This is how our spy can look:

```
package net.lkrnac.blog.testing.mockbeanv2.aoptesting;

import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Primary;
import org.springframework.context.annotation.Profile;
import org.springframework.stereotype.Service;

import lombok.Getter;
import net.lkrnac.blog.testing.mockbeanv2.beans.AddressDao;
import net.lkrnac.blog.testing.mockbeanv2.beans.AddressService;

@Primary
@Service
@Profile("UserService-aop-test")
public class AddressServiceSpy extends AddressService{
    @Getter
    private AddressService spyDelegate;
```

```
    private AddressService spyDelegate;

    @Autowired
    public AddressServiceSpy(AddressDao addressDao) {
        super(null);
        spyDelegate = Mockito.spy(new AddressService(addressDao));
    }

    public String getAddressForUser(String userName){
        return spyDelegate.getAddressForUser(userName);
    }
}
```

As we can see, this spy is very similar to **`AddressDaoMock`** . But in this case, the real Bean is using constructor injection to autowire its dependency. Therefore, we'll need to define non-default constructors and do constructor injection also. But we won't pass injected dependencies into parent constructor.

To enable spying on a real object, we construct a new instance with all the dependencies, wrap it into a Mockito spy instance, and store it in the **`spyDelegate`** property. We expect to call **`getAddressForUser()`** during testing, therefore we delegate this call to **`spyDelegate`** . This property can be accessed in tests via the getter defined by Lombok's **`@Getter`** annotation.

The test itself will look like this:

```
@ActiveProfiles({"UserService-aop-test", "aop"})
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(AopApplication.class)
public class UserServiceAopITest {
    @Autowired
    private UserService userService;

    @Autowired
    private AddressService addressService;

    @Test
    public void testGetUserDetails() {
        // GIVEN
        AddressServiceSpy addressServiceSpy = (AddressServiceSpy) addressService;

        // WHEN
        String actualUserDetails = userService.getUserDetails("john");

        // THEN
        Assert.assertEquals("User john, 3 Dark Corner", actualUserDetails);
        Mockito.verify(addressServiceSpy.getSpyDelegate()).getAddressForUser("john");
    }
}
```

It is very straight forward. Profile **`UserService-aop-test`** ensures that **`AddressServiceSpy`** will be scanned. Profile **`aop`** ensures same for the **`AddressLogger`** aspect. When we autowire the testing object **`UserService`** and its dependency **`AddressService`** , we know that we can cast it to **`AddressServiceSpy`** and verify the call on its **`spyDelegate`** property after calling the testing method.

Faking a Spring Bean Proxied by Spring AOP

It is obvious that delegating calls into Mockito mocks or spies complicates the testing. These patterns are often overkill

if we simply need to fake the logic. We can use a fake in that case:

```
@Primary
@Repository
@Profile("AddressService-aop-fake-test")
public class AddressDaoFake extends AddressDao{
    public String readAddress(String userName) {
        return userName + "'s address";
    }
}
```

and use it for testing this way:

```
@ActiveProfiles({"AddressService-aop-fake-test", "aop"})
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(AopApplication.class)
public class AddressServiceAopFakeITest {
    @Autowired
    private AddressService addressService;

    @Test
    public void testGetAddressForUser() {
        // GIVEN - Spring context

        // WHEN
        String actualAddress = addressService.getAddressForUser("john");

        // THEN
        Assert.assertEquals("john's address", actualAddress);
    }
}
```

I don't think this test needs explanation.

Source code for these examples is hosted on Github.

Like This Article? Read More From DZone

Spring Core Skills: 5000 Ways to Build and Wire Your Spring Beans [Video]

Playing Around With Spring Bean Configuration

An Interview Question on Spring Singletons

Free DZone Refcard
Quarkus

Topics: SPRING , JAVA , MOCKING , SPRING TUTORIAL , BEANS , SPRING BEANS

Published at DZone with permission of Lubos Krnac , DZone MVB. See the original article here.

Using Java Optional Vs. Vavr Option

by Yuri Mednikov · Sep 27, 19 · Java Zone · Presentation

When it comes to Java, there are so many Options...

Today, I would like to discuss an essential Java topic – the usage of `Optional` class — and compare it with an alternative from the Vavr library. Optional was initially introduced in Java 8 and defined as “a container object which may or may not contain a non-null value.”

You may also like: 26 Reasons Why Using Optional Correctly Is Not Optional

Developers utilize Optionals in order to avoid null checking in places when code execution leads to "not a result" but also to a *null* value, and it can, in this regard, result in a `NullPointerException` . In such cases, Optional offers us some fancy functionality, but not all of it was introduced in the 8th release, some features require Java 11. Another way to handle these issues is with Vavr’s `Option` class.

In this post, we learn how to use Java’s `Optional` class, and then compare it to Vavr’s `Option` class. Note: this code requires Java 11 + and was tested with Vavr *0.10.2*.

Let's get started.

Introducing Java Optional

The concept of Optional is not new and has been already implemented in functional programming languages like Haskell or Scala. It proves to be very useful when modeling cases when a method call could return an unknown value or a value that does not exist (e.g. nulls). Let's see how to handle it.

Creation of Optional

First things first, we need to obtain an Optional's instance. There are several ways to do it – and moreover, we also can create an *empty* Optional. Check out the first method – creating from value, it is pretty straightforward:

```
Optional<Integer> four = Optional.of(Integer.valueOf(4));
if (four.isPresent){
System.out.println("Hoorayy! We have a value");
} else {
System.out.println("No value");
}
```

We build an `Optional` from an `Integer` value of 4, meaning that there always should be a value and it cannot be null, but this is just an example. We check an existence or absence of value with the `ifPresent()` method. You can note that `four` is not an `Integer`; it is a container *that holds integer inside*. When we are sure that the value is inside, we can "unpack" it with the `get()` method. Ironically, if we use `get()` without checking, we can end it with `NoSuchElementException`.

Another way to obtain an `Optional` is using streams. Several terminal stream's methods return Optionals, so we can manipulate them and check their existence or absence, for example:

- `findAny`
- `findFirst`
- `max`
- `min`
- `reduce`

Check out the code snippet below:

```
Optional<Car> car = cars.stream().filter(car->car.getId().equalsIgnoreCase(id)).findFirst();
```

The next option is to create `Optional` from code that *can potentially produce null*, e.g. from `Nullable`:

```
Optional<Integer> nullable = Optional.ofNullable(client.getRequestData());
```

Finally, we can create an *empty* `Optional`:

```
Optional<Integer> nothing = Optional.empty();
```

How to Use Optional

As long as we obtained `Optional`, we can use it. One of the most widespread cases is using it in Spring repositories to find one record by Id, so we can build our logic on `Optional` and avoid null checking (btw, Spring also supports Vavr Options). Let's say we have a book repository and want to find one book.

```
Optional<Book> book = repository.findOne("some id");
```

First of all, we can execute some logic, in this case, if the book *is presented*. We did it with `if-else` in the previous

section, but we don't need to: `Optional` provides us a method that accepts a *Consumer* with the object:

```
repository.findOne("some id").ifPresent(book -> System.out.println(book));
```

Or, we can make this even simpler; we can write the same with method references:

```
repository.findOne("some id").ifPresent(System.out::println);
```

If we don't have a book in the repository, we can provide an alternative callback with the `ifPresentOrElseGet` method:

```
repository.findOne("some id").ifPresentOrElseGet(book->{
// if value is presented
}, ()->{
// if value is absent
});
```

Alternatively, we can get another value if the result of the operation is not presented:

```
Book result = repository.findOne("some id").orElse(defaultBook);
```

However, with Optionals, we need to remember possible drawbacks. In the last example, we "guarantee" ourselves that we could obtain a book anyway; either it presents the underlying repository or comes from `orElse`. But what if this default value is not *constant*, but also requires some complex method? First, Java **anyway** evaluates `findOne`. Then, it has to process the `orElse` method. Yes, if it is just a default constant value, it is OK, but it can be, as I said before, time-consuming as well.

Another Example

Let's create a simple example to check how to practically use `Optional` and `Option` classes. We would have a `CarRepository` that would find a car based on the supplied ID (e.g. on the license plate number). Then, we would see how to manipulate Optionals and Options.

First, Let's Add Some Code

Start with the POJO class `Car`. It follows the *immutable* pattern, so all fields are final and we have only getters without setters. All data is supplied during initialization.

```
public class Car {

    private final String name;
    private final String id;
    private final String color;

    public Car (String name, String id, String color){
        this.name = name;
        this.id = id;
        this.color = color;
    }

    public String getId(){
        return id;
```

```
    }

    public String getColor() {
        return color;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "Car "+name+" with license id "+id+" and of color "+color;
    }
}
```

The second thing is to create the `CarRepository` class. It requires two options for finding car by Id — using the old way with a possible null result and using `Optional`, as we do in Spring repositories.

```
public class CarRepository {

    private List<Car> cars;

    public CarRepository(){
       getSomeCars();
    }

    Car findCarById(String id){
        for (Car car: cars){
            if (car.getId().equalsIgnoreCase(id)){
                return car;
            }
        }
        return null;
    }

    Optional<Car> findCarByIdWithOptional(String id){
        return cars.stream().filter(car->car.getId().equalsIgnoreCase(id)).findFirst();
    }

    private void getSomeCars(){
        cars = new ArrayList<>();
        cars.add(new Car("tesla", "1A9 4321", "red"));
        cars.add(new Car("volkswagen", "2B1 1292", "blue"));
        cars.add(new Car("skoda", "5C9 9984", "green"));
        cars.add(new Car("audi", "8E4 4321", "silver"));
        cars.add(new Car("mercedes", "3B4 5555", "black"));
        cars.add(new Car("seat", "6U5 3123", "white"));
    }
}
```

Note that we also populate our repository with some mock cars during initialization, so we don't have any underlying database. We need to avoid complexities, so let's concentrate on `Optional` and `Option`, not on repositories.

Finding Cars With Java Optional

Create a new test with JUnit:

```
@Test
void getCarById(){
    Car car = repository.findCarById("1A9 4321");
    Assertions.assertNotNull(car);
    Car nullCar = repository.findCarById("M 432 KT");
    Assertions.assertThrows(NullPointerException.class, ()->{
        if (nullCar == null){
            throw new NullPointerException();
        }
    });
}
```

The above code snippet demonstrates the old way. We found a car with the Czech license plate **1A9 4321** and checked that it exists. Then we found a car that is absent, as it has a Russian license plate and we only have Czech ones in our repository. It is *null*, so it may lead to `NullPointerException`.

Next, let's move to Java Optionals. The first step is to obtain an `Optional` instance from the repository using a dedicated method that returns `Optional`:

```
@Test
void getCarByIdWithOptional(){
    Optional<Car> tesla = repository.findCarByIdWithOptional("1A9 4321");
    tesla.ifPresent(System.out::println);
}
```

In this case, we use the `findCarByIdWithOptional` method and then print a car (if it presents). If you run it, you will get the following output:

```
Car tesla with license id 1A9 4321 and of color red
```

But what if we don't have that special method in our code? In this situation, we can obtain `Optional` from a method that could potentially return a null value. It is called `nullable`.

```
Optional<Car> nothing = Optional.ofNullable(repository.findCarById("5T1 0965"));
Assertions.assertThrows(NoSuchElementException.class, ()->{
    Car car = nothing.orElseThrow(()->new NoSuchElementException());
});
```

In this code snippet, we found another way. We create `Optional` from `findCarById` and that can return *null* if no car is found. We manually use the `orElseThrow` method to throw a `NoSuchElementException` when the desired car with the license plate **5T1 0965** is present. Another situation is to use `orElse` with a default value if the requested data is not available in the repository:

```
Car audi = repository.findCarByIdWithOptional("8E4 4311")
```

```
            .orElse(new Car("audi", "1W3 4212", "yellow"));
    if (audi.getColor().equalsIgnoreCase("silver")){
      System.out.println("We have silver audi in garage!");
    } else {
      System.out.println("Sorry, there is no silver audi, but we called you a taxi");
}
```

Ok, we don't have the silver Audi in our garage, so we have to call a taxi!

Finding Cars With the Vavr Option

Vavr `Option` is another way to handle these tasks. First, install Vavr in your project with dependency management (I use Maven):

```
<dependency>
   <groupId>io.vavr</groupId>
   <artifactId>vavr</artifactId>
   <version>0.10.2</version>
</dependency>
```

In a nutshell, Vavr has similar APIs to create `Option` instances. We can create `Option` from nullable, as shown here:

```
Option<Car> nothing = Option.of(repository.findCarById("T 543 KK"));
```

Or we can create an *empty* container with the `none` static method:

```
Option<Car> nullable = Option.none();
```

Also, there is a way to create `Option` from Java `Optional` ! Take a look at the code snippet below:

```
Option<Car> result = Option.ofOptional(repository.findCarByIdWithOptional("5C9 9984"));
```

With Vavr `Option` , we can use the same API as with `Optional` to accomplish the previously mentioned tasks. For example, we can set a default value in a similar manner:

```
Option<Car> result = Option.ofOptional(repository.findCarByIdWithOptional("5C9 9984"));
Car skoda = result.getOrElse(new Car("skoda", "5E2 4232", "pink"));
System.out.println(skoda);
```

Or we can throw an exception based on an absence of requested data:

```
Option<Car> nullable = Option.none();
Assertions.assertThrows(NoSuchElementException.class, ()->{
nullable.getOrElseThrow(()->new NoSuchElementException());
});
```

Alternatively, we can perform an action when data is unavailable:

```
nullable.onEmpty(()->{
```

```
nullable.onEmpty(()->{
///runnable
});
```

What if we need to perform an action based on a presence of data, as we did with Optional's `ifPresent` ? We can do this in several ways. There is an equal method to `isPresent` in `Optional` that in `Option` is called `isDefined` :

```
if (result.isDefined()){
// do something
}
```

However, we use `Option` to get rid of if-else constructs. Can we float in the same way as with `Optional` ? We can perform operations based on existence with `peek` :

```
result.peek(val -> System.out.println(val)).onEmpty(() -> System.out.println("Result is missed"));
```

Also, there are some other very useful methods in Vavr `Option` that can make your code even more functional than with the built-in `Optional` class. So, I encourage you to take some time and explore Vavr Option javadocs and experiment with these APIs. I will go ahead and note some cool features like `map` , `narrow` , `isLazy` , and `when` that you definitely need to check out.

Also, Vavr `Option` is a part of the Vavr family and is heavily integrated with other Vavr classes, and making such a comparison with Optional in the absence of such classes is not correct. So, I also plan to write other posts on Vavr topics like Try, Collections, and Streams. Stay tuned!

Conclusion

In this post, we talked about the `Optional` class in Java. The concept of `Optional` is not something new and has been already implemented in other functional programming languages like Haskell and Scala. It proves to be very useful when modeling cases when a method call could return an unknown value or a value that does not exist (e.g. nulls). Then, we explored its APIs and created some examples finding cars and manipulating results with `Optional` logic. And finally, we discovered an alternative to `Optional` – Vavr's `Option` and described its methods as well.

Hope you enjoyed! Be sure to leave thoughts or questions in the comments.

Further Reading

26 Reasons Why Using Optional Correctly Is Not Optional

How to Be More Functional in Java With Vavr

A Look at Java Optionals

Like This Article? Read More From DZone

Functional Programming in Java 8 (Part 2): Optionals

Lifting Functions to Work With Java Monads

Understanding flatMap

Free DZone Refcard

Quarkus

Topics: JAVA, VAVR, JAVA 11, FUNCTIONAL PROGRAMMING, TUTORIAL, OPTION, OPTIONAL

A Bootiful Podcast: Nicolai Parlog on Java Modularity

by Joshua Long MVB · Sep 27, 19 · Java Zone · Interview

Hi, Spring fans! In this installment, Josh Long (@starbuxman) talks to Manning's *The Java Module System* author Nicolai Parlog (@nipafx) about Java modularity. Let's get started.

A Bootiful Podcast

Nicolai Parlog on Java modularity Compartir

Further Reading

Java Modularity: A Personal Journey

[DZone Refcard] Patterns of a Modular Architecture

Java 9 Modular Development: Parts 1 & 2

Like This Article? Read More From DZone

Architecture of Spring Framework: Modularity and Spring Modules

Java 9 Module Services

Migrating a Spring Boot App to Java 9 (Part 2): Modules

Free DZone Refcard
Quarkus

Topics: JAVA, MODULARITY, MODULE, JAVA 9, JAVA 13, PODCAST, SPRING