Mockito

You are already familiar with unit tests and their main characteristics: they are small, fast, and we use them to test a unit of code in isolation.

However, sometimes it is not easy to comply with these criteria, especially if your code unit under test depends on external resources, such as files, databases, or web services.

There are several techniques that help you isolate a unit of code from its dependencies, and one of them is the use of so-called **mock objects**.

A mock object is a simulated object that imitates the behavior of a real object in controlled ways. In unit tests, you can use mock objects instead of real objects to interact with the code under test and verify the result of such interaction.

You can create mock objects manually or use a mocking framework that will do the heavy lifting for you.

Mockito

One of the most popular mocking frameworks for Java is the Mockito framework. It has a simple and convenient API, allows you to write clean and concise tests, and can be used together with JUnit.

If you have a Maven project, add the following dependencies:

```
<dependencies>
  <dependency>
<!--      JUnit5 -->
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>5.7.1</version>
      <scope>test</scope>
  </dependency>

  <dependency>
<!--      Mockito -->
      <groupId>org.mockito</groupId>
      <artifactId>mockito-core</artifactId>
      <version>3.11.2</version>
      <scope>test</scope>
  </dependency>
</dependencies>
```

For a Gradle project, add the following:

```
dependencies {
// JUnit5
   testImplementation 'org.junit.jupiter:junit-jupiter:5.7.1'
// Mockito
   testImplementation 'org.mockito:mockito-core:3.11.2'
}
```

You are already familiar with unit tests and their main characteristics: they are small, fast, and we use them to test a unit of code in isolation. However, sometimes it is not easy to comply with these criteria, especially if your code unit under test depends on external resources, such as files, databases, or web services. There are several techniques that help you isolate a unit of code from its dependencies, and one of them is the use of so-called **mock objects**.
A mock object is a simulated object that imitates the behavior of a real object in controlled ways. In unit tests, you can use mock objects instead of real objects to interact with the code under test and verify the result of such interaction. You can create mock objects manually or use a mocking framework that will do the heavy lifting for you.

Mockito

One of the most popular mocking frameworks for Java is the Mockito framework. It has a simple and convenient API, allows you to write clean and concise tests, and can be used together with JUnit.
First, let's see how to add Mockito to your project.
If you have a Maven project, add the following dependencies:

```
<dependencies>
   <dependency>
<!--       JUnit5 -->
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>5.7.1</version>
      <scope>test</scope>
   </dependency>

   <dependency>
<!--       Mockito -->
      <groupId>org.mockito</groupId>
      <artifactId>mockito-core</artifactId>
      <version>3.11.2</version>
      <scope>test</scope>
   </dependency>
</dependencies>
```

For a Gradle project, add the following:

```
dependencies {
// JUnit5
   testImplementation 'org.junit.jupiter:junit-jupiter:5.7.1'
// Mockito
   testImplementation 'org.mockito:mockito-core:3.11.2'
}
```

Mockito developers regularly release new versions, so always check for the most recent version available.

If your class has a dependency...

```
import java.math.BigDecimal;
import java.math.RoundingMode;

public class FXConverter {
   private final RemoteFXRateService remoteFXRateService;

   public FXConverter(RemoteFXRateService remoteFXRateService) {
      this.remoteFXRateService = remoteFXRateService;
   }

   public BigDecimal convert(String source, String target, String input) {
      try {
         String response = remoteFXRateService.getRate(source, target);
         BigDecimal rate = new BigDecimal(response);
         BigDecimal amount = new BigDecimal(input);

         return amount.multiply(rate).setScale(2, RoundingMode.HALF_UP);
      } catch (IllegalStateException | IllegalArgumentException ex) {
         return new BigDecimal("-1.00");
      }
   }
}
```

we have a class named FXConverter responsible for converting an amount in one currency to the proportionate amount in another currency.

For this purpose, it uses the convert method that accepts the source and the target currency codes, together with the input amount, as String and returns the converted amount as BigDecimal.

FXConverter has a RemoteFXRateService dependency, which has the getRate method that sends a request to a foreign exchange web service to retrieve the actual exchange rate between the source currency and the target currency and return it as String.

This method can throw an IllegalArgumentException if it receives a negative response from the web service for the given pair of currency codes passed as arguments. It can also throw an IllegalStateException if the web service does not respond for any reason.

In case of an error, convert returns the BigDecimal equivalent of "-1.00", and in the case of success, it returns the converted amount.

Now that it's done, we want to write unit tests to make sure that our FXConverter works as intended. As you can see, we cannot simply create an instance of the FXConverter class because it has the RemoteFXRateService class as a dependency, so we have to instantiate that class first.

First, RemoteFXRateService can also have some dependencies, and those dependencies can have dependencies too... In other words, the number of objects we have to instantiate can grow like a snowball

Second, we want to test our FXConverter class in isolation with all its dependencies tested beforehand. Also, we want to test our class in controlled conditions and avoid hard reliance on external data.

Fortunately, Mockito allows us to avoid these difficulties and provides an easy-to-use API to create mock objects and define their behavior.

Creating mock objects
Let's set up unit tests for our FXConverter class. There are two ways to instantiate a mock object of the RemoteFXRateService class. The first one is with the use of mock, a static method that creates a default mock of the specified class.

```
import static org.mockito.Mockito.mock;

class FXConverterTest {

    private RemoteFXRateService service = mock(RemoteFXRateService.class);

    private FXConverter converter = new FXConverter(service);

}
```

The second way is using annotations. If you want to use this method, you have to add another dependency to your project:

```
testImplementation 'org.mockito:mockito-junit-jupiter:3.11.2'
```

Then you can set up the required classes using the appropriate annotations:

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
class FXConverterTest {

   @Mock
   private RemoteFXRateService service;

   private FXConverter converter;

   @BeforeEach
   void setup() {
      converter = new FXConverter(service);
   }
}
```

If you are unsure which of these two methods to choose, keep in mind that while the mock method requires less code, it only supports raw classes, so you cannot use it to instantiate, for example, List<String>; it can only generate a raw List.

Defining the behavior

Next, we need to define the behavior of the mock object for different test cases.

We want to test the convert method using a number of edge cases, including the situations in which the getRate method throws an IllegalArgumentException or anIllegalStateException, and when it returns the exchange rate.

For this purpose, Mockito has other static methods: when and thenReturn. They define the answer that must be returned when the specified method of the mock object is invoked with the specified arguments.

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;
```

```
class FXConverterTest {

   private RemoteFXRateService service = mock(RemoteFXRateService.class);

   private FXConverter converter = new FXConverter(service);

   @Test
   @DisplayName("Given 100.00 USD, when convert to USD, then return
100.00")
   void test1() {
      when(service.getRate("USD", "USD")).thenReturn("1.0000");

      BigDecimal result = converter.convert("USD", "USD", "100.00");

      assertEquals("100.00", result.toString());
   }
}
```

Here we specify that if the getRate method of the service object is called with the same two arguments, "USD" and "USD", then the method must return "1.0000", because the exchange rate of USD to USD, in any case, will be equal to 1.0. Then we call the convert method and assert that the returned value is effectively equal to the value passed as an argument.

```
FXConverterTest > Given 100.00 USD, when convert to USD, then return 100.00
PASSED
```

Let's run it for two different currencies:

```
@Test
@DisplayName("Given 100.00 USD, when convert to EUR, then return 84.97")
void test2() {
   when(service.getRate("USD", "EUR")).thenReturn("0.8497");

   BigDecimal result = converter.convert("USD", "EUR", "100.00");

   assertEquals("84.97", result.toString());
}
```

Again, the test is passed successfully. But what if we call the convert method with a different pair of arguments, for example, "USD" and "GBP"? The test will fail because we did not define any answer for this combination of arguments for getRate

Argument matchers

Mockito provides default behavior for all created mock objects, so any unspecified method calls will return the following values:

- null for objects
- 0 for numbers
- false for boolean
- empty collections for collections

Mockito has the ArgumentMatchers class whose static methods give us great flexibility in specifying arguments: we can define the behavior of a method depending on whether the argument is equal or not equal to a certain value, is of a certain type, is null or not null, matches to a certain pattern, etc

official documentation

```
import static org.mockito.ArgumentMatchers.anyString;
import static org.mockito.ArgumentMatchers.contains;
import static org.mockito.ArgumentMatchers.endsWith;
import static org.mockito.ArgumentMatchers.eq;
import static org.mockito.ArgumentMatchers.startsWith;
import static org.mockito.Mockito.when;

class FXConverterTest {

   private RemoteFXRateService service = mock(RemoteFXRateService.class);

   private FXConverter converter = new FXConverter(service);

   @Test
   void test3() {
      // 1st arg is "USD" and 2nd arg is any string that contains "coin"
      when(service.getRate(eq("USD"), contains("coin")))
            .thenReturn("0.0000");

      // both 1st arg and 2nd arg is any string
      when(service.getRate(anyString(), anyString())).thenReturn("42");

      // 1st arg is any string that starts with "US"
      // and 2nd arg is any string that ends with "BP"
      when(service.getRate(startsWith("US"), endsWith("BP")))
            .thenReturn("0.7266");
   }

}
```

Common argument matchers are:

Method	Description
any()	Matches anything (returns null)
anyInt()	Any non-null Integer (returns 0)
anyString()	Any non-null String (returns empty string)
eq()	The argument of the corresponding type that is equal to the given value (returns 0)

If you are using argument matchers, all arguments have to be provided by matchers:
when(mock.someMethod(anyInt(), eq("second argument"))).thenReturn(42);
// correct because eq() is also an argument matcher

when(mock.someMethod(anyInt(), "second argument")).thenReturn(42);
// incorrect because "second argument" is not an argument matcher.
We will get InvalidUseOfMatchersException

Throwing exceptions
Finally, let's see how to change the behavior of any mocked method to make it throw an exception. For this purpose, Mockito has another static method called thenThrow (do not forget to import it):

```
@Test
@DisplayName("Given any args, when service throws exception, then return
-1.00")
void test4() {
   when(service.getRate(anyString(), anyString()))
         .thenThrow(new IllegalStateException());

   BigDecimal result =
         converter.convert("USD", "EUR", "100.00");

   assertEquals("-1.00", result.toString());
}
```

Whenever the getRate method throws an IllegalStateException, the convert method returns the BigDecimal equivalent of "-1.00". If we run this test, we will see that it is successfully passed:

```
FXConverterTest> Given any args, when service throws exception, then return
-1.00 PASSED
```