

Mockito Hands-On Exercises

Exercise 1: Mocking and Stubbing Scenario:

You need to test a service that depends on an external API. Use Mockito to mock the external API and stub its methods.

Steps:

1. Create a mock object for the external API.
2. Stub the methods to return predefined values.
3. Write a test case that uses the mock object.

Solution Code: `import static org.mockito.Mockito.*;`

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

`import org.mockito.Mockito;`

`public class MyServiceTest`

`{`

`@Test public void testExternalApi()`

`{ ExternalApi mockApi = Mockito.mock(ExternalApi.class);`

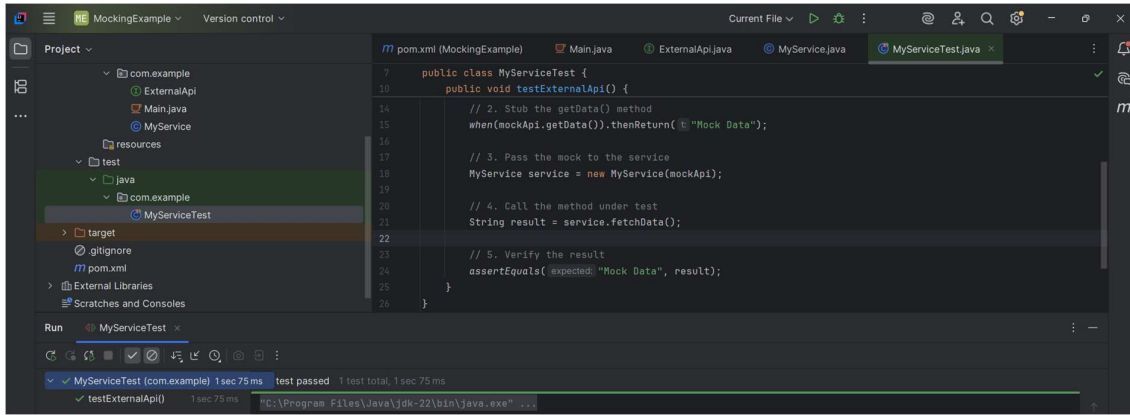
`when(mockApi.getData()).thenReturn("Mock Data");`

`MyService service = new MyService(mockApi);`

`String result = service.fetchData();`

`assertEquals("Mock Data", result); }`

Output:



Exercise 2: Verifying Interactions

Scenario: You need to ensure that a method is called with specific arguments.

Steps:

1. Create a mock object.
2. Call the method with specific arguments.
3. Verify the interaction.

Solution Code:

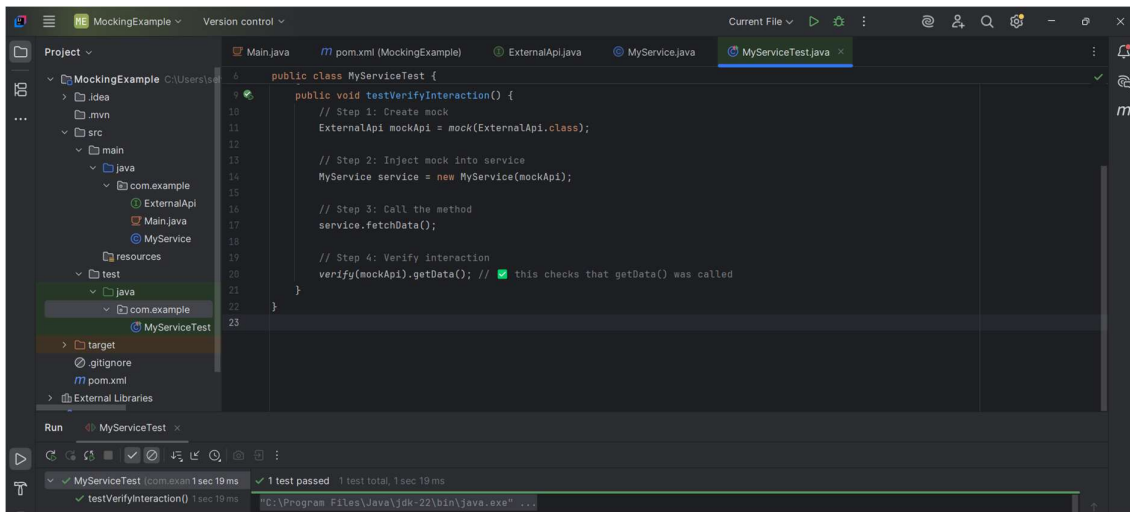
```
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

public class MyServiceTest {

    @Test public void testVerifyInteraction() {

        ExternalApi mockApi = Mockito.mock(ExternalApi.class);
        MyService service = new MyService(mockApi);
        service.fetchData();
        verify(mockApi).getData(); } }
```

Output:



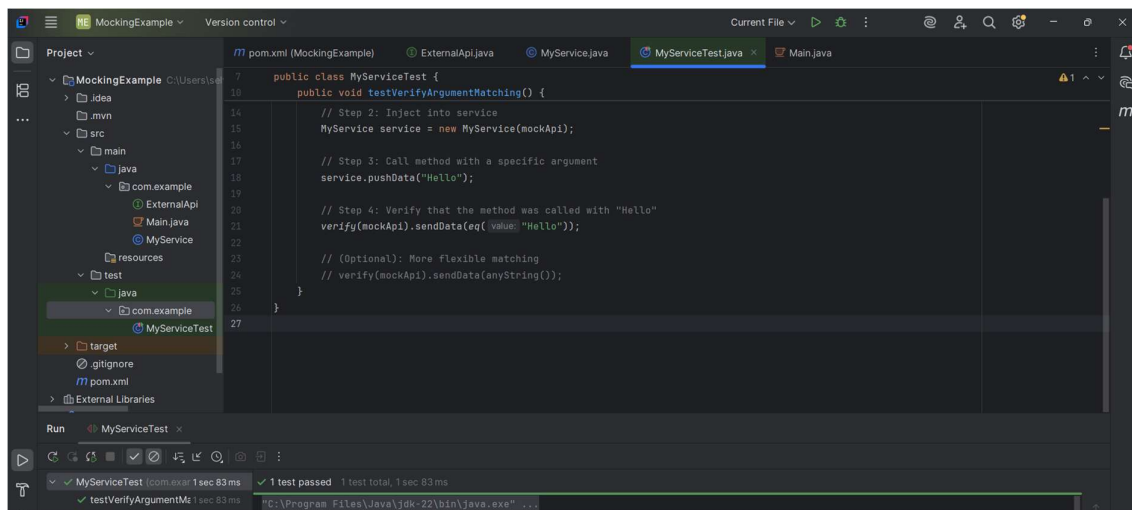
Exercise 3: Argument Matching

Scenario: You need to verify that a method is called with specific arguments.

Steps:

1. Create a mock object.
2. Call the method with specific arguments.
3. Use argument matchers to verify the interaction

Output:



```
1 public class MyServiceTest {
2     public void testVerifyArgumentMatching() {
3         // Step 1: Create a mock object
4         MyService mockApi = Mockito.mock(MyService.class);
5
6         // Step 2: Inject into service
7         MyService service = new MyService(mockApi);
8
9         // Step 3: Call method with a specific argument
10        service.pushData("Hello");
11
12        // Step 4: Verify that the method was called with "Hello"
13        verify(mockApi).sendData(eq("Hello"));
14
15        // (Optional): More flexible matching
16        // verify(mockApi).sendData(anyString());
17    }
18 }
```

Run MyServiceTest

MyServiceTest [com.example] 1 sec 83 ms 1 test passed 1 test total, 1 sec 83 ms

testVerifyArgumentMatching [com.example] 1 sec 83 ms

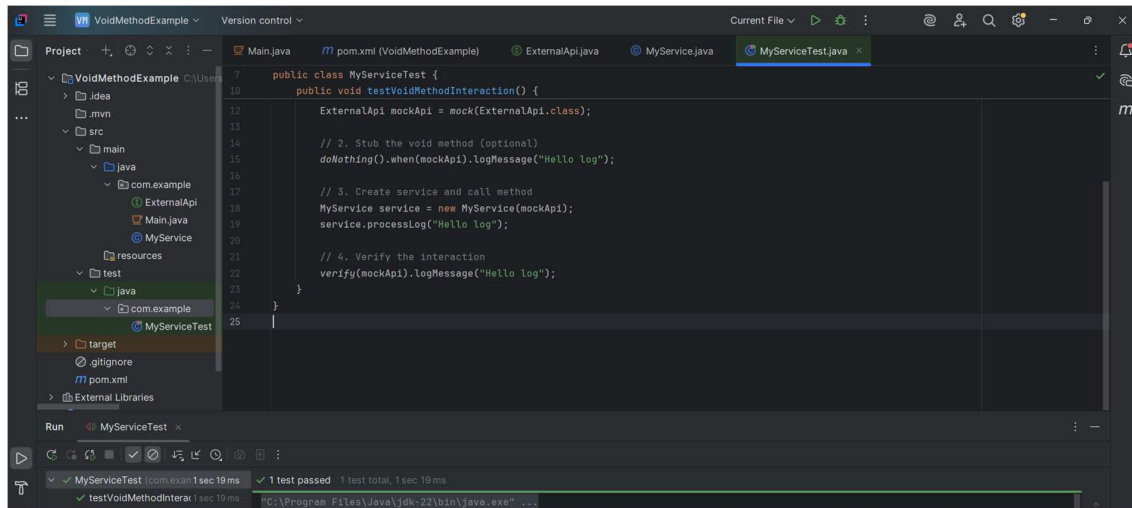
Exercise 4: Handling Void Methods

Scenario: You need to test a void method that performs some action.

Steps:

1. Create a mock object.
2. Stub the void method.
3. Verify the interaction.

Output:



```
7 public class MyServiceTest {
8     public void testVoidMethodInteraction() {
9
10         ExternalApi mockApi = mock(ExternalApi.class);
11
12         // 2. Stub the void method (optional)
13         doNothing().when(mockApi).logMessage("Hello log");
14
15         // 3. Create service and call method
16         MyService service = new MyService(mockApi);
17         service.processLog("Hello log");
18
19         // 4. Verify the interaction
20         verify(mockApi).logMessage("Hello log");
21     }
22 }
23
24
25
```

Run: MyServiceTest

✓ MyServiceTest (com.example) 1 sec 19 ms ✓ 1 test passed 1 test total, 1 sec 19 ms

testVoidMethodInterac 1 sec 19 ms

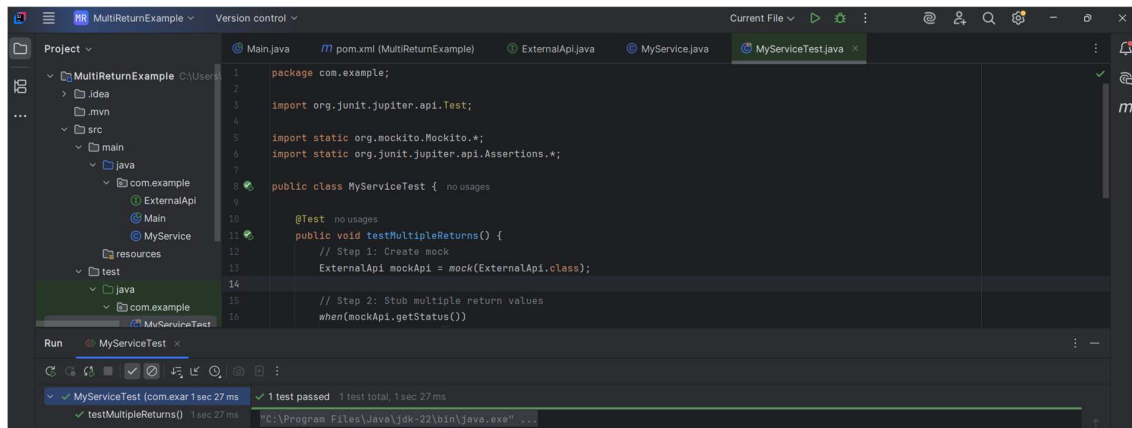
Exercise 5: Mocking and Stubbing with Multiple Returns

Scenario: You need to test a service that depends on an external API with multiple return values.

Steps:

1. Create a mock object for the external API.
2. Stub the methods to return different values on consecutive calls.
3. Write a test case that uses the mock object.

Output:



The screenshot shows an IDE window for a project named 'MultiReturnExample'. The project structure on the left includes 'src/main/java/com/example' with files 'ExternalApi.java', 'Main.java', and 'MyService.java'. The 'test' directory contains 'com/example' with 'MyServiceTest.java'. The editor displays the code for 'MyServiceTest.java':

```
1 package com.example;
2
3 import org.junit.jupiter.api.Test;
4
5 import static org.mockito.Mockito.*;
6 import static org.junit.jupiter.api.Assertions.*;
7
8 public class MyServiceTest {
9     // no usages
10
11     @Test
12     // no usages
13     public void testMultipleReturns() {
14         // Step 1: Create mock
15         ExternalApi mockApi = mock(ExternalApi.class);
16
17         // Step 2: Stub multiple return values
18         when(mockApi.getStatus())
```

The Run window at the bottom shows the test execution results:

```
Run MyServiceTest
MyServiceTest (com.example) 1 sec 27 ms ✓ 1 test passed 1 test total, 1 sec 27 ms
testMultipleReturns() 1 sec 27 ms ✓
```

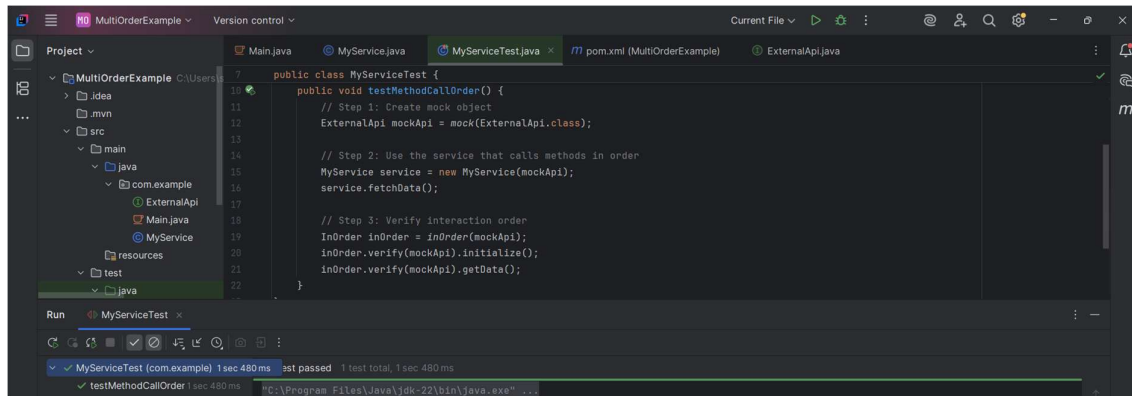
Exercise 6: Verifying Interaction Order

Scenario: You need to ensure that methods are called in a specific order.

Steps:

1. Create a mock object.
2. Call the methods in a specific order.
3. Verify the interaction order.

Output:



The screenshot shows an IDE window for a project named 'MultiOrderExample'. The 'Project' view on the left shows the directory structure: 'src' contains 'main' (with 'com.example' package containing 'ExternalApi', 'Main.java', and 'MyService') and 'test' (with 'java' package). The 'Main.java' file is open, showing the following code:

```
public class MyServiceTest {  
    public void testMethodCallOrder() {  
        // Step 1: Create mock object  
        ExternalApi mockApi = mock(ExternalApi.class);  
  
        // Step 2: Use the service that calls methods in order  
        MyService service = new MyService(mockApi);  
        service.fetchData();  
  
        // Step 3: Verify interaction order  
        InOrder inOrder = inOrder(mockApi);  
        inOrder.verify(mockApi).initialize();  
        inOrder.verify(mockApi).getData();  
    }  
}
```

The 'Run' view at the bottom shows the test results for 'MyServiceTest (com.example)'. The test 'testMethodCallOrder' passed successfully, taking 1 sec 480 ms. The total test time is 1 sec 480 ms.

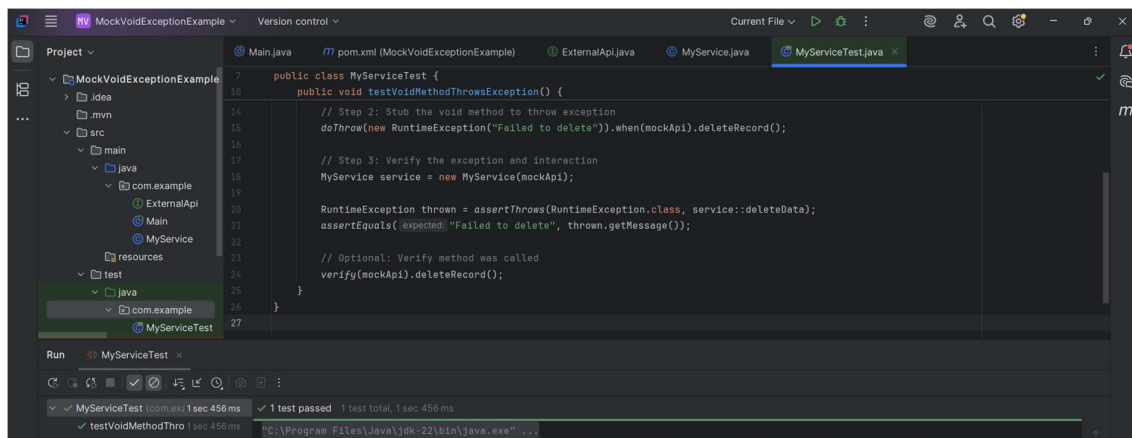
Exercise 7: Handling Void Methods with Exceptions

Scenario: You need to test a void method that throws an exception.

Steps:

1. Create a mock object.
2. Stub the void method to throw an exception.
3. Verify the interaction

Output:



```
7 public class MyServiceTest {
8     public void testVoidMethodThrowsException() {
9
10        // Step 2: Stub the void method to throw exception
11        doThrow(new RuntimeException("Failed to delete")).when(mockApi).deleteRecord();
12
13        // Step 3: Verify the exception and interaction
14        MyService service = new MyService(mockApi);
15
16        RuntimeException thrown = assertThrows(RuntimeException.class, service::deleteData);
17        assertEquals("expected: 'Failed to delete', thrown.getMessage());
18
19        // Optional: Verify method was called
20        verify(mockApi).deleteRecord();
21    }
22 }
23
24
25
26
27
```

Run MyServiceTest

MyServiceTest [com.example] 1 sec 456 ms ✓ 1 test passed 1 test total, 1 sec 456 ms

testVoidMethodThro 1 sec 456 ms