

# 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.

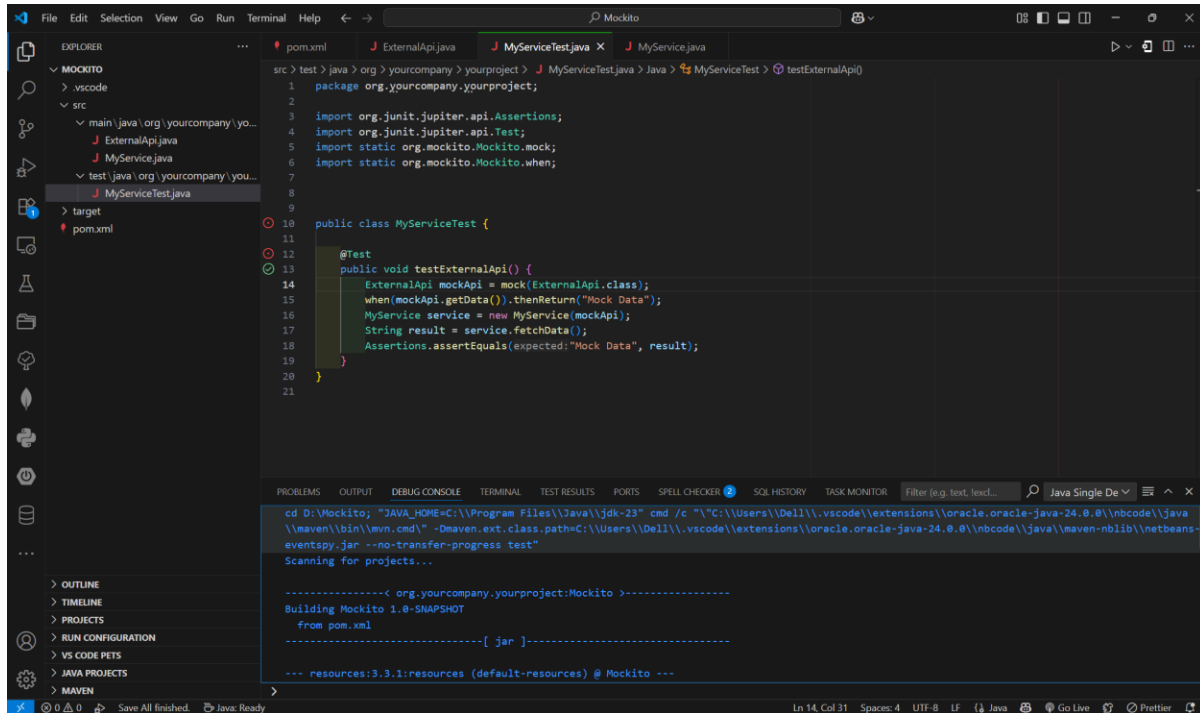
Steps:

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);
    }
}
```



## 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.*;
```

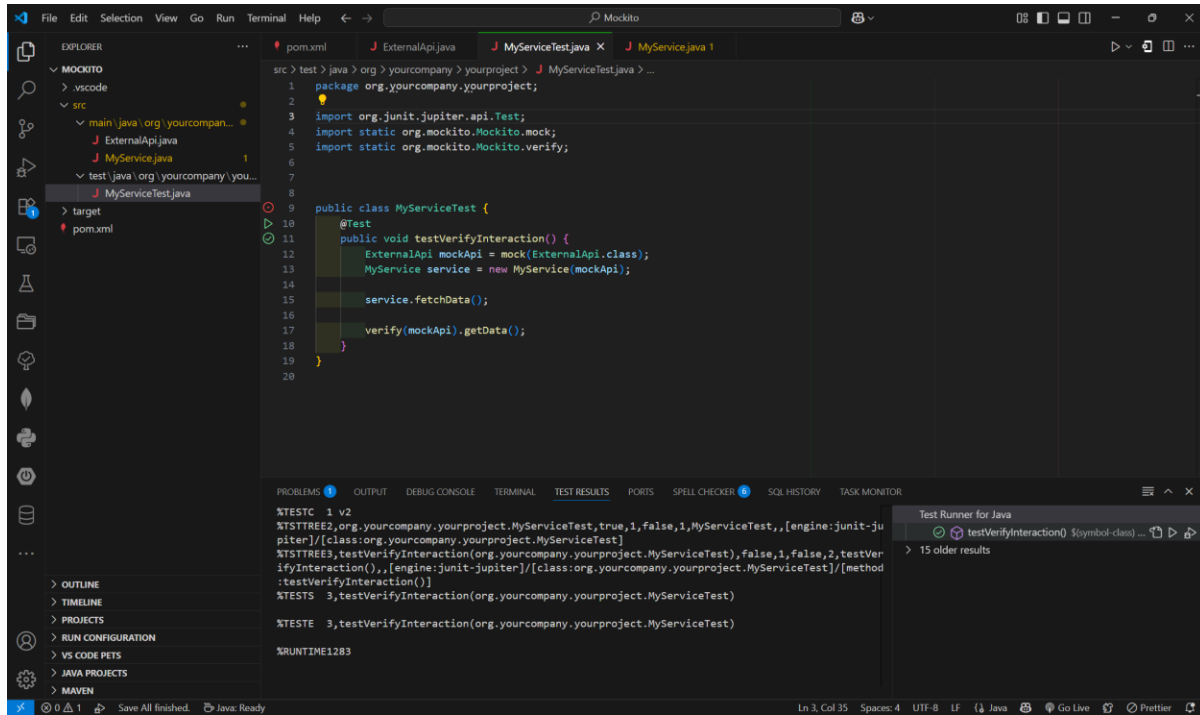
```
import org.junit.jupiter.api.Test;
```

```
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();
    }
}

```



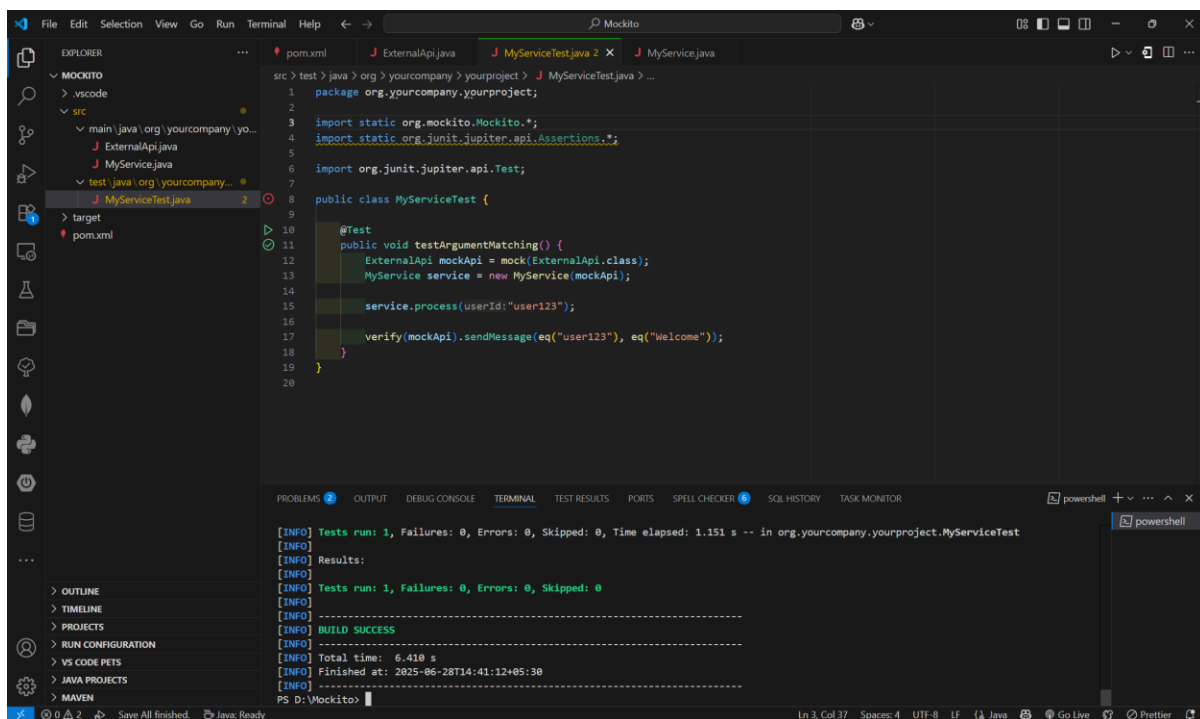
### Exercise 3: Argument Matching

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.



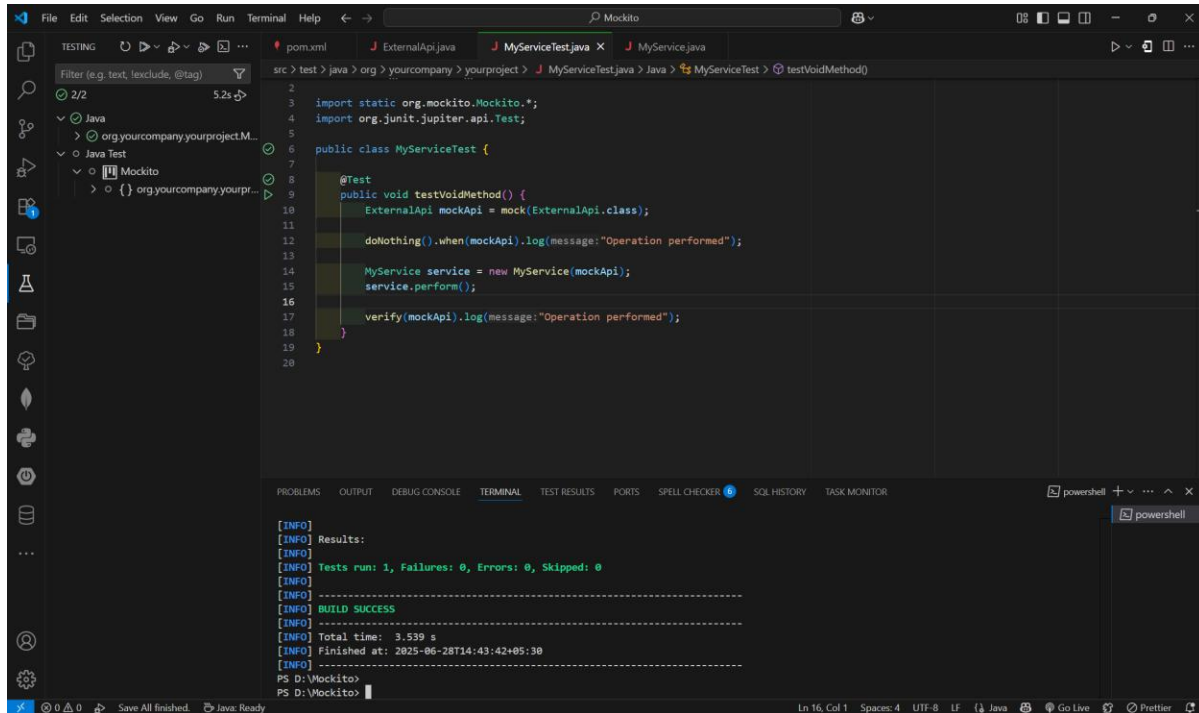
## Exercise 4: Handling Void Methods

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.



## 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.

```

src > test > java > org > yourcompany > yourproject > J MyServiceTest.java > ...
4  import static org.junit.jupiter.api.Assertions.*;
5
6  import org.junit.jupiter.api.Test;
7
8  public class MyServiceTest {
9
10     @Test
11     public void testMultipleReturnValues() {
12         ExternalApi mockApi = mock(ExternalApi.class);
13
14         // Stub with multiple return values
15         when(mockApi.getStatus())
16             .thenReturn("Loading")
17             .thenReturn("Completed");
18
19         MyService service = new MyService(mockApi);
20         String result = service.checkStatusTwice();
21
22         assertEquals(expected:"Loading then Completed", result);
23     }
24 }
25
[INFO] Results:
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO] BUILD SUCCESS
[INFO] Total time: 3.466 s
[INFO] Finished at: 2025-06-28T14:44:58+05:30
[INFO]
PS D:\Mockito>

```

## 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.

```

src > test > java > org > yourcompany > yourproject > J MyServiceTest.java > ...
1  package org.yourcompany.yourproject;
2
3  import static org.mockito.Mockito.*;
4  import org.junit.jupiter.api.Test;
5  import org.mockito.InOrder;
6
7  public class MyServiceTest {
8
9     @Test
10     public void testInteractionOrder() {
11         ExternalApi mockApi = mock(ExternalApi.class);
12
13         MyService service = new MyService(mockApi);
14         service.run();
15
16         // Verify the order of method calls
17         InOrder inOrder = inOrder(mockApi);
18         inOrder.verify(mockApi).initialize();
19         inOrder.verify(mockApi).fetchData();
20         inOrder.verify(mockApi).cleanup();
21     }
22 }
23
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.155 s -- in org.yourcompany.yourproject.MyServiceTest
[INFO] Results:
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO] BUILD SUCCESS
[INFO] Total time: 6.855 s
[INFO] Finished at: 2025-06-28T14:50:11+05:30
[INFO]
PS D:\Mockito>

```

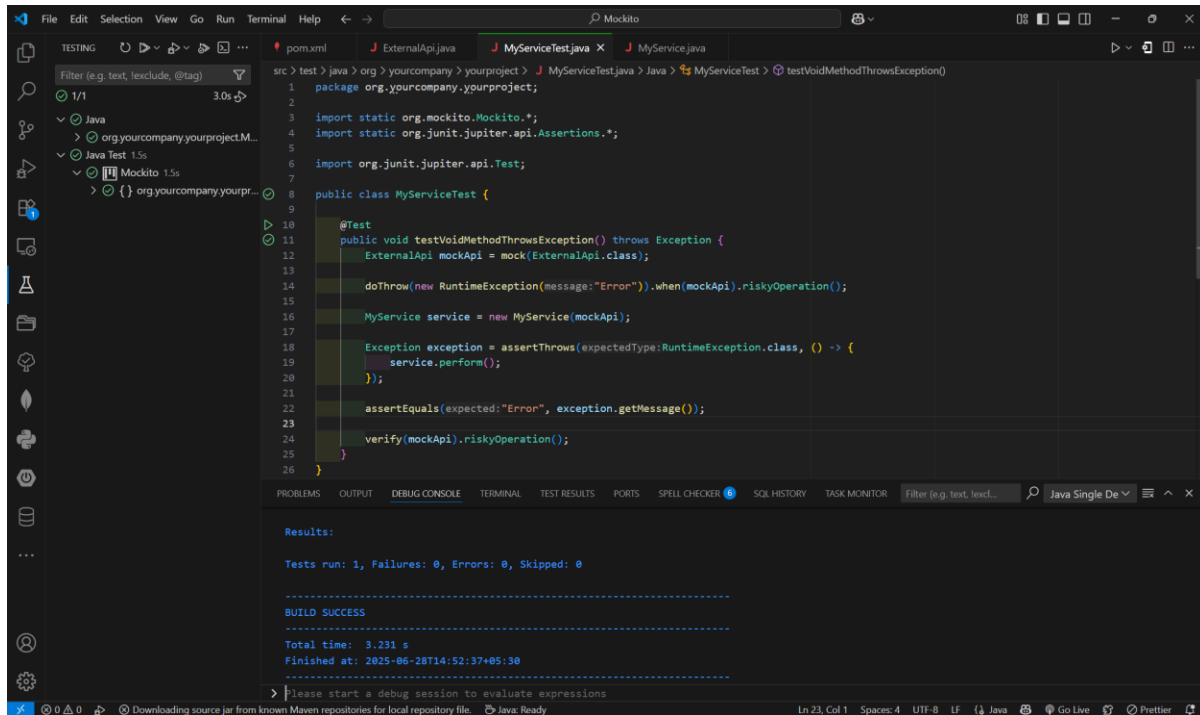
## Exercise 7: Handling Void Methods with Exceptions

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.



The screenshot shows an IDE window with a Java test class `MyServiceTest.java` and its execution results in the terminal.

**Code in `MyServiceTest.java`:**

```
1 package org.yourcompany.yourproject;
2
3 import static org.mockito.Mockito.*;
4 import static org.junit.jupiter.api.Assertions.*;
5
6 import org.junit.jupiter.api.Test;
7
8 public class MyServiceTest {
9
10     @Test
11     public void testVoidMethodThrowsException() throws Exception {
12         ExternalApi mockApi = mock(ExternalApi.class);
13
14         doThrow(new RuntimeException(message: "Error")).when(mockApi).riskyOperation();
15
16         MyService service = new MyService(mockApi);
17
18         Exception exception = assertThrows(RuntimeException.class, () -> {
19             service.perform();
20         });
21
22         assertEquals(expected: "Error", exception.getMessage());
23
24         verify(mockApi).riskyOperation();
25     }
26 }
```

**Terminal Output:**

```
Results:
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

-----
BUILD SUCCESS
-----
Total time: 3.231 s
Finished at: 2025-06-28T14:52:37+05:30
-----
```

The terminal also shows a message: "Please start a debug session to evaluate expressions".