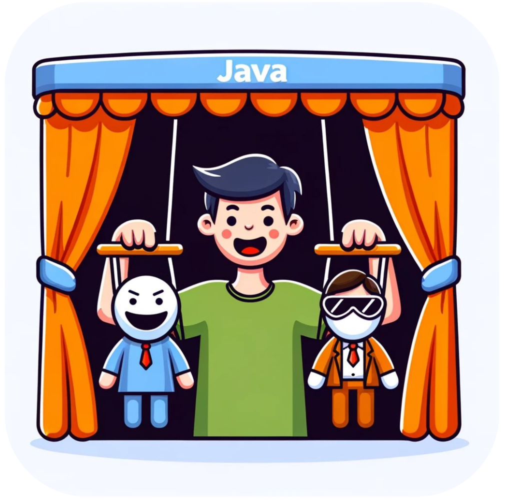
Mock Objects:
Using and Creating Mocks with Mockito



Creating Mocks with Mockito

Setting Up Mockito in Maven

```xml
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
</dependency>
```

Creating Mocks with Mockito using Mockito.mock()

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

public class MockitoMockExample {

    @Test
    public void testUsingMockitoMock() {
        // Creating a mock object of List interface
        List<String> mockedList = mock(List.class);

        // Use the mock
        mockedList.add("one");
        mockedList.clear();

        // Verify the behavior
        verify(mockedList).add("one");
        verify(mockedList).clear();
    }
}
```

Creating Mocks with Mockito using @Mock

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import java.util.List;
import static org.mockito.Mockito.*;

public class MockitoAnnotationExample {

    @Mock
    List<String> mockedList;

    @BeforeEach
    public void setup() {
        // Initialize the mock object
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void testUsingMockAnnotation() {
        // Use the mock
        mockedList.add("one");
        mockedList.clear();

        // Verify the behavior
        verify(mockedList).add("one");
        verify(mockedList).clear();
    }
}
```

Creating Mocks integration with JUnit 5

```java
@ExtendWith(MockitoExtension.class)
public class MockitoJUnit5Example {

    @Mock
    List<String> mockedList;

    @InjectMocks
    MyClass myClass;

    @Test
    public void testUsingExtendWith() {
        // Use the mock
        myClass.performOperation();

        // Verify the behavior
        verify(mockedList).add("one");
    }

    public static class MyClass {
        List<String> list;

        public void performOperation() {
            list.add("one");
        }
    }
}
```

Using Mocks with Mockito

Setting Return Values

```java
import static org.mockito.Mockito.*;

// Assume Calculator is a class with a method add
Calculator calculator = mock(Calculator.class);

// Setting up the mock to return 30 when add(10, 20) is called
when(calculator.add(10, 20)).thenReturn(30);
```

Verifying Interactions

```java
import static org.mockito.Mockito.*;

// Assume Printer is a class with a method print
Printer printer = mock(Printer.class);

// some method execution...

// Verifying that print was called
verify(printer, times(1)).print();
```

Resetting Mocks

```java
import static org.mockito.Mockito.*;

// Assume Calculator is a class with a method add
Calculator calculator = mock(Calculator.class);

// some method execution...

// Resetting the mock
reset(calculator);
```

Argument Matchers

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

List<String> mockedList = mock(List.class);

// Using standard matchers
when(mockedList.get(anyInt())).thenReturn("element");
when(mockedList.contains(eq("element"))).thenReturn(true);


// Using the mock
assertEquals("element", mockedList.get(0));
assertTrue(mockedList.contains("element"));
assertTrue(mockedList.contains("elemental"));
```

List of Argument Matchers

- **any()** - matches any object, regardless of its type.
  - `when(mockedList.get(anyInt())).thenReturn("element");`
- **anyInt(), anyDouble(), anyLong(),** etc. - matches any value of the specific primitive type.
  - `when(mockedList.get(anyInt())).thenReturn("element");`
- **eq()** - matches an argument that is equal to a specified value.
  - `when(mockedList.contains(eq("specificElement"))).thenReturn(true);`
- **isNull(), isNotNull()** - matches if the argument is null or not null, respectively.
  - `when(mockedList.add(isNull())).thenReturn(false);`
- **same()** - matches if the argument is the same object as specified.
  - `when(mockService.process(same(expectedObject))).thenReturn("processed");`
- **argThat()** - matches if the argument meets the conditions specified in a custom ArgumentMatcher.
  - `when(mockedList.contains(argThat(isValidCustomMatcher()))).thenReturn(true);`
- **anyListOf(Class<T>), anySetOf(Class<T>)**, etc. - matches any list, set, or other collection of a specific type.
  - `when(mockService.process(anyListOf(String.class))).thenReturn("processed");`
- **refEq()** - matches an argument that is equal to a specified object, comparing fields of the objects.
  - `when(mockService.update(refEq(expectedObject))).thenReturn("updated");`
- **lt(), gt(), leq(), geq()** - matches arguments less than, greater than, less than or equal to, or greater than or equal to a specified value.
  - `when(mockedList.get(lt(5))).thenReturn("less than 5");`
- **matches()** - matches if the argument string matches a specified regular expression.
  - `when(mockService.validate(matches("\\d+"))).thenReturn(true);`
- **contains(), startsWith(), endsWith()** - matches if the argument string contains, starts with, or ends with a specified substring.
  - `when(mockService.respond(contains("keyword"))).thenReturn("contains keyword");`
- **byteThat(), charThat(), shortThat()**, etc. - matches a byte, char, short, etc., based on a condition defined by an ArgumentMatcher.
  - `when(mockService.process(byteThat(byteConditionMatcher()))).thenReturn("processed");`