**Junit**

JUnit is a widely-used **unit testing framework for Java programming**, designed to help developers test individual units of their code (like methods or classes) to ensure they behave as expected. It's an integral part of Test-Driven Development (TDD) and ensures that your code is reliable, maintainable, and bug-free.

**Key Features of JUnit:**

1. **Annotations:** Simplify writing and organizing tests.
   * Example: @Test, @BeforeEach, @AfterEach, @BeforeAll, and @AfterAll.
2. **Assertions:** Verify the expected outputs using methods like assertEquals, assertTrue, and assertNotNull.
3. **Parameterized Tests:** Run the same test with multiple sets of input data.
4. **Integration:** Works seamlessly with build tools like Maven and Gradle.
5. **Test Suites:** Group multiple tests together to execute them as a batch.
6. **Reporting:** Generate test results and reports for debugging and performance evaluation.

**WHY?**

1. **Early Bug Detection:** Identify bugs during development, saving time and effort.
2. **Code Quality:** Ensure your code behaves as intended under various conditions.
3. **Regression Testing:** Prevent previously solved issues from reappearing after changes.
4. **Automation:** Automate testing, making it repeatable and efficient.
5. **Encourages TDD:** Supports Test-Driven Development by enabling developers to write tests before writing the code.

**Mockito:**

Mockito is a popular **Java mocking framework** used for unit testing. It allows developers to create mock objects, which simulate the behaviour of real objects in controlled test environments. With Mockito, you can test functionality without depending on external resources such as databases, APIs, or file systems.

**Why Mockito?**

* **Mocking Dependencies**: Simulate dependencies to isolate the code being tested.
* **Verify Interactions**: Ensure methods on mock objects are called with expected arguments.
* **Behaviour Stubbing**: Define the behaviour of mock objects for specific inputs.
* **Simplify Unit Tests**: Avoid complex setups of actual objects or systems.

Mockito is particularly useful in **unit testing** when you need to focus on a single class or method without relying on the functionality of other components.

**Key Concepts of Mockito:**

**1. Mocking:**

Mock objects mimic the behaviour of real objects without requiring actual implementation. Mockito provides the mock () method to create mock objects.

Example:

java

import static org.mockito.Mockito.\*;

import java.util.List;

public class MockingExample {

public static void main(String[] args) {

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

when(mockList.size()).thenReturn(5); // Stub a method

System.out.println(mockList.size()); // Prints 5

}

}

**2. Stubbing:**

Define the behavior of mock objects using when and thenReturn. If the method is called with the specified arguments, the stubbed response is returned.

Example:

java

import static org.mockito.Mockito.\*;

public class StubbingExample {

public static void main(String[] args) {

Calculator calculator = mock(Calculator.class);

when(calculator.add(2, 3)).thenReturn(5);

System.out.println(calculator.add(2, 3)); // Prints 5

}

}

interface Calculator {

int add(int a, int b);

}

**3. Verification:**

Mockito can verify if methods on mock objects were called with specific arguments and how many times.

Example:

java

import static org.mockito.Mockito.\*;

public class VerificationExample {

public static void main(String[] args) {

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

mockList.add("Hello");

verify(mockList).add("Hello"); // Verify that add() was called with "Hello"

verify(mockList, times(1)).add("Hello"); // Verify it was called exactly once

}

}

**4. Spy:**

A **spy** is a real object wrapped by Mockito. It lets you track interactions and override specific behaviors while using the actual implementation for other methods.

Example:

java

import static org.mockito.Mockito.\*;

import java.util.ArrayList;

public class SpyExample {

public static void main(String[] args) {

ArrayList<String> list = new ArrayList<>();

List<String> spyList = spy(list);

spyList.add("Hello");

spyList.add("World");

System.out.println(spyList.size()); // Prints 2

verify(spyList).add("Hello"); // Verify "Hello" was added

}

}

**5. Argument Matchers:**

Mockito provides matchers like any() and eq() to test methods that take dynamic arguments.

Example:

java

import static org.mockito.Mockito.\*;

public class ArgumentMatcherExample {

public static void main(String[] args) {

Calculator calculator = mock(Calculator.class);

when(calculator.add(anyInt(), eq(3))).thenReturn(6);

System.out.println(calculator.add(2, 3)); // Prints 6

}

}

**6. Annotations:**

Mockito provides annotations like @Mock, @InjectMocks, and @Spy to simplify the mocking process.

Example:

java

import org.mockito.Mock;

import org.mockito.InjectMocks;

import org.mockito.MockitoAnnotations;

import static org.mockito.Mockito.\*;

public class AnnotationExample {

@Mock

private Calculator calculator;

@InjectMocks

private MathService mathService;

public static void main(String[] args) {

AnnotationExample example = new AnnotationExample();

MockitoAnnotations.openMocks(example); // Initialize mocks

when(example.calculator.add(2, 3)).thenReturn(5);

System.out.println(example.mathService.addNumbers(2, 3)); // Prints 5

}

}

class Calculator {

public int add(int a, int b) {

return a + b;

}

}

class MathService {

private Calculator calculator;

public int addNumbers(int a, int b) {

return calculator.add(a, b);

}

}

**Features of Mockito:**

* No need for boilerplate code for mocking objects.
* Verifies interactions, ensuring tested code interacts correctly with its dependencies.
* Supports annotations to reduce manual mock setup.
* Flexible argument matchers for testing dynamic inputs.
* Works seamlessly with frameworks like Spring Boot and JUnit.

**Common Mockito Annotations:**

* @Mock: Creates a mock object for the specified class.
* @Spy: Wraps an actual object and tracks interactions.
* @InjectMocks: Automatically injects mocks into the specified object.
* @Captor: Captures arguments passed to a mock method.

Mockito simplifies unit testing and enables developers to isolate their code from external dependencies. Would you like me to help you set up Mockito for your project, or dive deeper into advanced mocking techniques? Let me know!