JUNIT5

Shristi Technology Labs

Contents

Introduction

- JUnit is a framework for implementing testing in Java.
- Provides a simple way to explicitly test specific areas of a program
- Is used to test a hierarchy of program code either singularly or as multiple units.
- Promotes the idea of first testing then coding
- Setup test data for a unit which defines ***what the expected output is and then code until the tests pass.***

Why Testing Framework

- is beneficial as it forces to explicitly declare the expected results of specific program execution routes.

- practice of *"test a little, code a little, test a little, code a little..."*
  - increases programmer productivity and stability of program code
  - reduces time spent debugging.

- By having a set of tests that test all the core components of the project
  - it is possible to modify specific areas of the project
  - see the effect of the modifications by the results of the test
  - side-effects can be quickly realized.

JUnit 5

- Supports Java 8
- Has new annotations and extensions
- JUnit 5 = JUnit platform + JUnit Jupiter + Junit Vintage

JUnit Architecture

JUnit Platform

- Helps to launch junit tests, IDEs, build tools or plugins
- Defines the TestEngine API for developing new testing frameworks that runs on the platform.
- Provides a Console Launcher to launch the platform from the command line and build plugins for Gradle and Maven.

JUnit Jupiter

- Has new programming and extension models for writing tests.
- Has new junit annotations and TestEngine implementation to run tests

JUnit Vintage

- Provides a TestEngine for running JUnit 3 and JUnit 4 written tests on the JUnit 5 platform.

Environment Setup

- Use Maven or gradle as build tool
- Add dependencies
  - Jupiter Engine
  - Jupiter API
  - Platform Runner

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.0.0-M4</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.0.0-M4</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-runner</artifactId>
    <version>1.0.0-M4</version>
    <scope>test</scope>
</dependency>
```

Testcase

- A testcase is a part of the code which ensures that a method works as expected.

- must be at least two unit test cases for each requirement − one positive test and one negative test.

Steps for writing testcases

- The test method should be annotated with @Test
- Neither test classes nor test methods need to be public.

```
public double deposit(int amount) {
    System.out.println("depositing");
    return balance + amount;
}
```

Class under Test

```
@Test
protected void testDeposit() {
    assertEquals("Should be 8200", 8200, bank.deposit(1200), 1.0);
}
```

Test case

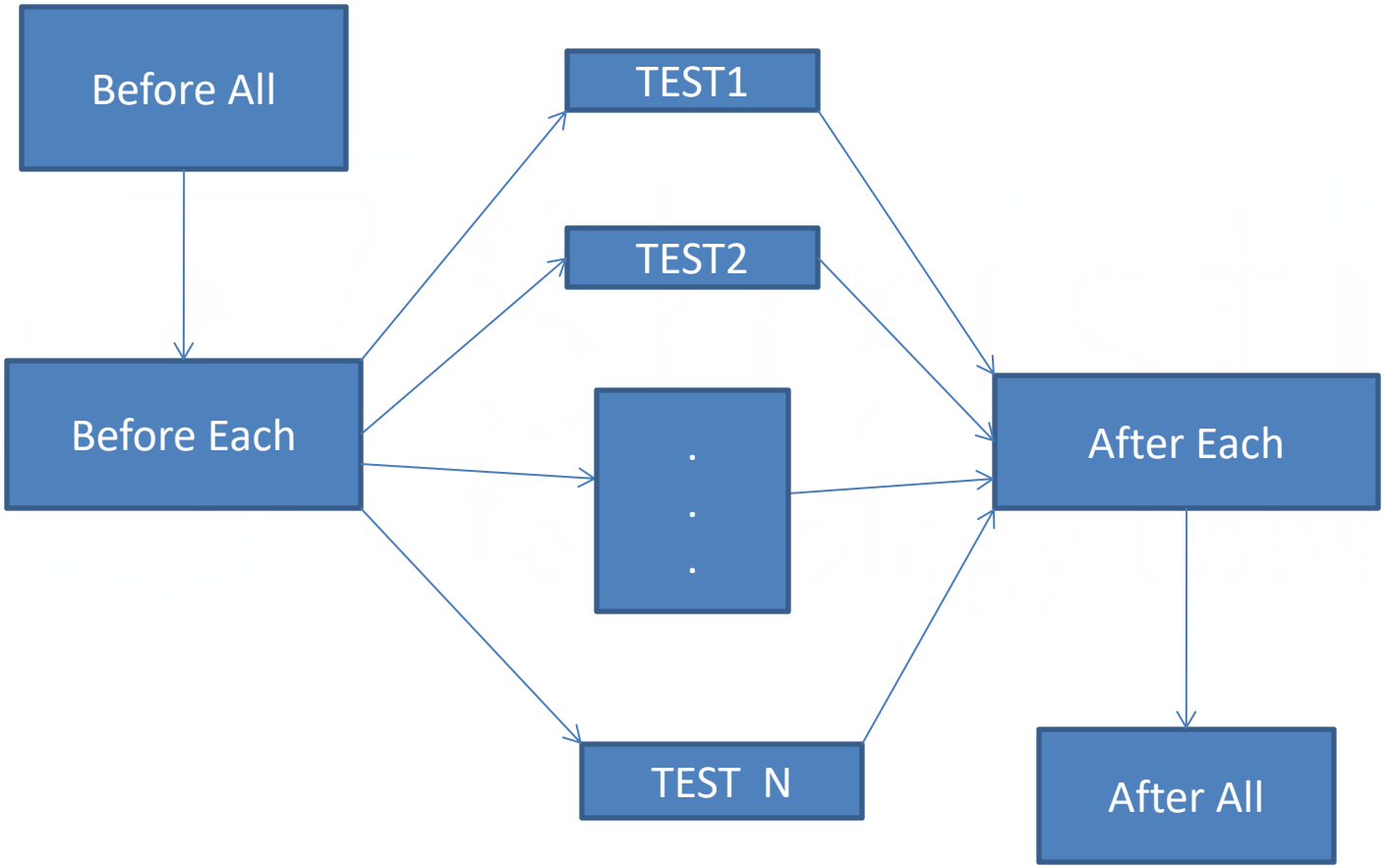
JUnit5 Annotations

ANNOTATION	DESCRIPTION
@Test	It is used to mark a method as JUnit test
@AfterEach	The annotated method will be run after each test method in the test class.
@BeforeEach	The annotated method will be run before each test method in the test class.
@BeforeAll	The annotated method will be run before all test methods in the test class. The method must be static.
@AfterAll	The annotated method will be run after all test methods in the test class. The method must be static.
@DisplayName	Used to provide any custom display name for a test class or test method
@Disable	It is used to disable or ignore a test class or method from test suite.
@Tag	Mark test methods or test classes with tags for test discovering and filtering

Annotations

```
@RunWith(JUnitPlatform.class)
public class SimpleTest {
    Bank bank = null;
    @BeforeAll
    static void init(){
        System.out.println("called before all test cases");
    }
    @BeforeEach
    void setUp(){
        bank = new Bank();
        System.out.println("called before each test");
    }
    @Test
    public void testAdd(){
        assertEquals("should be 8000",8000,bank.deposit(2000),1.0);
    }
    @AfterEach
    void teardown(){
        bank  = null;
        System.out.println("called after each test");
    }
    @AfterAll
    static void cleanup(){
        System.out.println("called after all test cases");
    }}
```

Test Instance Life Cycle



Assertions

- Assertions help in validating the expected output with actual output of a test case.
- Assertions are from ***org.junit.jupiter.Assertions***
- Third Party Assertion Libraries like `AssertJ, Hamcrest, Truth` are used to provide extra assert functions

Example

```
assertEqual(Object a, Object  b);
```

Assertions

assertNull(Object a); assertNotNull(Object a);	To check the initial value of the object/variable
assertTrue(condition); assertFalse(condition);	To test a boolean condition.
assertEqual(Object a, Object b);	To test equality of two objects.
assertEquals(Object a, Object b, String msg);	To test the equality of two objects with optional message
assertSame(Object a,Object b); **assertNotSame(Object a,Object b);**	To compare whether the reference of two object is same or different.
assertArrayEquals(expected,actual)	To compare arrays of same size.
fail(String msg);	To throw failure message.
assertTimeout(ofMinutes(int a), task);	Perform task within time a.
assertTimeoutPreemptivily(**ofMilli(int a), task);**	Simulates task that takes more than 'a' milli seconds
assertAll(String msg,assertfunctions…);	**Grouped assertion** →All assert statements are executed →Failures reported together. **Dependent assertion** →Within a code block, if an assertion fails the subsequent code in the same block will be skipped.

Example

```
Bank bank = null;
@BeforeEach
void setUp() {
    bank = new Bank();
    System.out.println("called before each test");
}
@Test   @DisplayName("Test Withdraw  - positive ")
public void testPassWithdraw() throws OutOfLimitsException {
    assertEquals("Incorrect", 5000, bank.withdraw(2000), 0.5);
}
@Test   @DisplayName("Test Withdraw  - negative ")
public void testFailWithdraw() throws OutOfLimitsException {
    assertEquals("Limit exceeded", 5000, bank.withdraw(12000), 0.5);
}
@AfterEach
void teardown() {
    System.out.println("called after each test");
    bank = null;    }
```

Checking Exceptions

- To test exceptions, use **`Assertions.assertThrows()`** method

```
public double withdraw(double amount) throws OutOfLimitsException{
    if(amount>2000){
        throw new OutOfLimitsException("out of limits");
    }
    return balance-amount;
}
```

```
@Test
void testExpection() {
    Assertions.assertThrows(OutOfLimitsException.class, () -> {
        bank.withdraw(9000);
    });
}
```

- If an exception is thrown, then the testcase will PASS.
- If an exception of a different type is thrown, the testcase will FAIL.(but allows exception of super types)

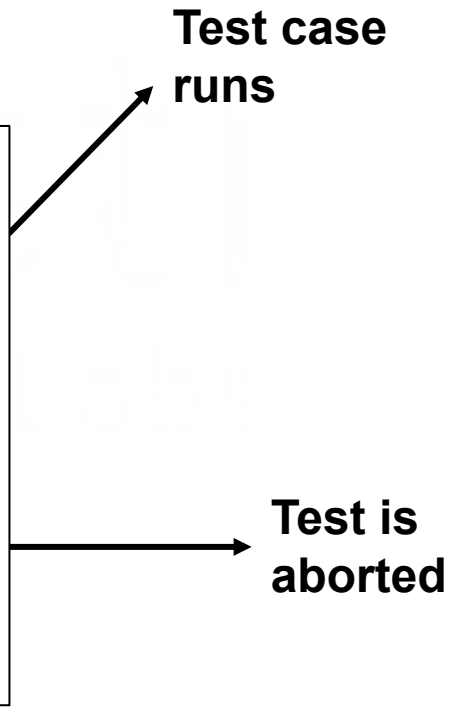
Assumptions

- Assumptions are static methods to support conditional test execution based on assumptions.
- A failed assumption will abort the test
- Assumptions are from ***`org.junit.jupiter.Assumptions`***
- Assumptions have two methods
  - ***`assumeTrue(Arguments);`***
  - ***`assumingThat(Arguments);`***

assumeTrue()

- This method validates that the given assumption to true
- If assumption is true – proceeds the test, else aborts the test.

```
@Test
void testOnOrders(){
    System.setProperty("env", "DEV");
    Assumptions.assumeTrue(System.getProperty("env").equals("DEV"));
    // true - so proceeds with the test
    assertSame("Hello", "Hello");
}
@Test
void testOnCart(){
    System.setProperty("env", "PROD");
    Assumptions.assumeTrue(System.getProperty("env").equals("DEV"));
    //false so the test will be aborted
    assertSame("Hello", "Hello");
}
```



assumeFalse()

- This method validates that the given assumption to false
- If assumption is false – proceeds the test, else aborts the test.

```
@Test
void testOnOrders(){
    System.setProperty("env", "DEV");
    Assumptions.assumeFalse(System.getProperty("env").equals("DEV"));
    // true - so the test will be aborted
    assertSame("Hello", "Hello");
}
@Test
void testOnCart(){
    System.setProperty("env", "PROD");
    Assumptions.assumeFalse(System.getProperty("env").equals("DEV"));
    //if false proceed, so proceeds with the test
    assertSame("Hello", "Hello");
}
```

Test is aborted

Test case runs

TestSuite

- TestSuite allows to run several tests spread into multiple test classes and different packages.
- The annotations used are
  - @SelectPackages
  - @SelectClasses

```
@RunWith(JUnitPlatform.class)
@SelectPackages("{com.training.usertests,com.training.ordertests}")
public class BankTest {
```

```
@RunWith(JUnitPlatform.class)
@SelectClasses({SimpleTest.class, UserTest.class})
public class BankTest {
```

Filtering Packages

@IncludePackage

- To perform testcases only for the classes in the subpackage

```
@RunWith(JUnitPlatform.class)
@SelectPackages("com.training.tests")
@IncludePackages("com.training.tests.usertests")
public class BankTest {
```

Only testcases inside usertests will be executed

@ExcludePackage

- To exclude any subpackage in the main package

```
@RunWith(JUnitPlatform.class)
@SelectPackages("com.training.tests")
@ExcludePackages("com.training.tests.usertests")
public class BankTest {
```

Testcases inside usertests will not be executed

Tagging

- The testcases and the methods can be tagged
- Use **@Tag** annotation used to tag a class or a test method
- Tags help to create multiple test plans for different environments, different use-cases or any specific requirement.

```
@Tag("simple")
public class SimpleTest {
    @Test
    @Tag("first")
    void testCalcInterest() {
    }
}
```

Filtering

- Filter the tagged tests by configuring the filters extension.

```
@Tag("development")
class UserTest {
    @Test
    @Tag("first")
    void testName() {
    }
}
@Tag("production")
class LoanTest {
    @Test
    @Tag("interest")
    void testCalcInterest() {
    }
    @Test
    @Tag("payment")
    void testPayment() {
    }
}
```

The tests with the tags in @IncludeTags annotation only will be executed

```
@RunWith(JUnitPlatform.class)
@SelectPackages("com.training.tests")
@IncludeTags("production")
public class BankTest {
```

Repeated Test

- **@RepeatedTest** provides the ability to repeat a test a specified number of times
- Has few placeholders as
  - **{displayName}** Display name
  - **{currentRepetition}** Current repetition count
  - **{totalRepetitions}** Total number of repetitions

```
@Test
@RepeatedTest(value = 4 , name="{displayName}..{currentRepetition}")
@DisplayName("Testing Withdraw method")
public void testWithdraw(){
    assertEquals("Incorrect",5000, bank.withdraw(2000),0.5);
}
@Test
@RepeatedTest(value = 10 , name="repeat deposit test{totalRepetitions} ")
public void testDeposit(){
    assertEquals("Incorrect",8200, bank.deposit(1200),1.0);
}
```

ParameterizedTest

- Parameterized tests helps to run a test multiple times with different arguments.
- The methods are annotated with @ParameterizedTest and they take arguments
- The values for the arguments are taken from a value source or method source or Enum source
- To perform parameterizedTest add the dependency junit-jupiter-params

```
<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-params -->
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
</dependency>
```

ParameterizedTest

Example – Runs twice with the values

```
@ParameterizedTest
@ValueSource(strings = {"Great day","Good day"})
void testGreet(String message) {
    //value for parameter is taken from ValueSource
    assertEquals(message, user.greet());
}
```

Summary

- Overview
- Architecture
- Environment Setup
- Annotations
- Writing Tests
- Assertions & Assumptions
- Handling Exception
- Test Suites
- Tagging and Filtering
- Repeated Tests

Thank You