JUNIT ADVANCED TESTING EXERCISE HANDS-ON

Exercise 1: Parameterized Tests

Scenario: You want to test a method that checks if a number is even. Instead of writing multiple test cases, you will use parameterized tests to run the same test with different inputs.

Code:

EvenCheckerTest.java

**package** TestSuit;

**import** org.junit.jupiter.api.BeforeEach;

**import** org.junit.jupiter.params.ParameterizedTest;

**import** org.junit.jupiter.params.provider.ValueSource;

**import** **static** org.junit.jupiter.api.Assertions.\*;

**public** **class** EvenCheckerTest {

**private** EvenChecker evenChecker;

**void** setUp() {

evenChecker = **new** EvenChecker();

}

@ParameterizedTest

@ValueSource(ints = {2, 4, 6, 8, 10, 100, 1000, -2, -4, 0})

**void** testIsEven\_WithEvenNumbers\_ShouldReturnTrue(**int** evenNumber) {

// Act & Assert

*assertTrue*(evenChecker.isEven(evenNumber),

"Expected " + evenNumber + " to be even");

}

}

Evenchecker.java

**package** TestSuit;

**public** **class** EvenChecker {

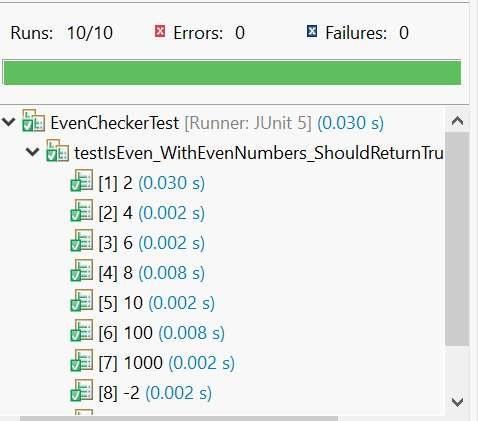
**public** **boolean** isEven(**int** number) {

**return** number % 2 == 0;

}

}

Output :



Exercise 2: Test Suites and Categories

Scenario: You want to group related tests into a test suite and categorize them.

Code :

Calculator.java

**package** TestSuit;

**public** **class** Calculator {

**public** **static** **int** add(**int** a, **int** b) {

**return** a + b;

}

**public** **static** **int** subtract(**int** a, **int** b) {

**return** a - b;

}

}

AdditionTest.java

**package** TestSuit;

//AdditionTest.java

**import** **static** org.junit.jupiter.api.Assertions.\*;

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

**class** AdditionTest {

@Test

**void** testAddition() {

Calculator calculator = **new** Calculator();

*assertEquals*(5, calculator.*add*(2, 3));

}

}

Subtraction.java

**package** TestSuit;

//SubtractionTest.java

**import** **static** org.junit.jupiter.api.Assertions.\*;

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

**class** SubtractionTest {

@Test

**void** testSubtraction()

{

Calculator calculator = **new** Calculator();

*assertEquals*(2, calculator.*subtract*(5, 3));

}

}

CalculatorTest.java

**package** TestSuit;

//CalculatorTestSuite.java

**import** org.junit.platform.suite.api.SelectClasses;

**import** org.junit.platform.suite.api.Suite;

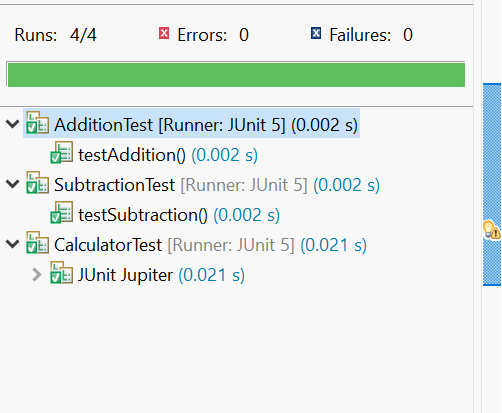
@Suite

@SelectClasses({AdditionTest.**class**, SubtractionTest.**class**})

**public** **class** CalculatorTest {

}

Output :



Exercise 3: Test Execution Order Scenario:

You want to control the order in which tests are executed.

Code:

Program.java

**package** TestSuit;

**public** **class** Program {

**public** **int** findSum(**int** a,**int** b)

{

**return** a+b;

}

**public** **int** findDiff(**int** a,**int** b)

{

**return** a-b;

}

**public** **int** findMult(**int** a,**int** b)

{

**return** a\*b;

}

}

OrderedTest.java

**package** TestSuit;

**import** **static** org.junit.Assert.*assertEquals*;

**import** org.junit.Test;

**import** org.junit.jupiter.api.MethodOrderer;

**import** org.junit.jupiter.api.Order;

**import** org.junit.jupiter.api.TestMethodOrder;

@TestMethodOrder(MethodOrderer.OrderAnnotation.**class**)

**public** **class** OrderedTest {

@Test

@Order(2)

**public** **void** testdiff()

{

Program obj=**new** Program();

*assertEquals*(2,obj.findDiff(4,2));

System.***out***.println("In testdiff");

}

@Test

@Order(1)

**public** **void** testmul()

{

Program obj=**new** Program();

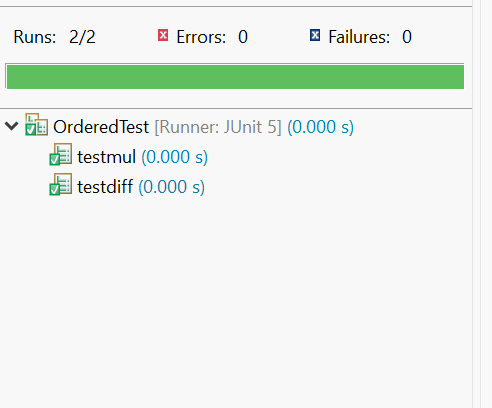
*assertEquals*(8,obj.findMult(4,2));

System.***out***.println("In testmul");

}

}

Output:



Exercise 4: Exception Testing

Scenario: You want to test that a method throws the expected exception

Code :

ExceptionThrowerTest.java

**package** TestSuit;

**import** org.junit.jupiter.api.BeforeEach;

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

**import** **static** org.junit.jupiter.api.Assertions.\*;

**public** **class** ExceptionThrowerTest {

**private** ExceptionThrower exceptionThrower;

@BeforeEach

**void** setUp() {

exceptionThrower = **new** ExceptionThrower();

}

// Tests for throwException method

@Test

**void** throwException\_WithNullInput\_ShouldThrowIllegalArgumentException() {

// Act & Assert

IllegalArgumentException exception = *assertThrows*(

IllegalArgumentException.**class**,

() -> exceptionThrower.throwException(**null**)

);

*assertEquals*("Input cannot be null", exception.getMessage());

}

@Test

**void** throwException\_WithEmptyInput\_ShouldThrowIllegalArgumentException() {

// Act & Assert

IllegalArgumentException exception = *assertThrows*(

IllegalArgumentException.**class**,

() -> exceptionThrower.throwException("")

);

*assertEquals*("Input cannot be empty", exception.getMessage());

}

@Test

**void** throwException\_WithWhitespaceInput\_ShouldThrowIllegalArgumentException() {

// Act & Assert

IllegalArgumentException exception = *assertThrows*(

IllegalArgumentException.**class**,

() -> exceptionThrower.throwException(" ")

);

*assertEquals*("Input cannot be empty", exception.getMessage());

}

@Test

**void** throwException\_WithInvalidNumberFormat\_ShouldThrowNumberFormatException() {

// Act & Assert

NumberFormatException exception = *assertThrows*(

NumberFormatException.**class**,

() -> exceptionThrower.throwException("abc")

);

*assertEquals*("Invalid number format: abc", exception.getMessage());

}

@Test

**void** throwException\_WithZero\_ShouldThrowArithmeticException() {

// Act & Assert

ArithmeticException exception = *assertThrows*(

ArithmeticException.**class**,

() -> exceptionThrower.throwException("0")

);

*assertEquals*("Cannot divide by zero", exception.getMessage());

}

@Test

**void** throwException\_WithValidInput\_ShouldNotThrowException() {

// Act & Assert - should not throw any exception

*assertDoesNotThrow*(() -> exceptionThrower.throwException("10"));

*assertDoesNotThrow*(() -> exceptionThrower.throwException("-5"));

*assertDoesNotThrow*(() -> exceptionThrower.throwException("100"));

}

// Tests for validateAge method

@Test

**void** validateAge\_WithNegativeAge\_ShouldThrowIllegalArgumentException() {

// Act & Assert

IllegalArgumentException exception = *assertThrows*(

IllegalArgumentException.**class**,

() -> exceptionThrower.validateAge(-1)

);

*assertEquals*("Age cannot be negative", exception.getMessage());

}

@Test

**void** throwException\_MultipleExceptionScenarios() {

// Test multiple exception scenarios

*assertThrows*(IllegalArgumentException.**class**,

() -> exceptionThrower.throwException(**null**));

*assertThrows*(IllegalArgumentException.**class**,

() -> exceptionThrower.throwException(""));

*assertThrows*(NumberFormatException.**class**,

() -> exceptionThrower.throwException("invalid"));

*assertThrows*(ArithmeticException.**class**,

() -> exceptionThrower.throwException("0"));

}

}

ExceptionThrower.java

**package** TestSuit;

**public** **class** ExceptionThrower {

/\*\*

\* Throws an exception based on the input parameter

\* **@param** input the input string to process

\* **@throws** IllegalArgumentException if input is null or empty

\* **@throws** NumberFormatException if input cannot be parsed as integer

\* **@throws** ArithmeticException if input is zero (division by zero simulation)

\*/

**public** **void** throwException(String input) {

**if** (input == **null**) {

**throw** **new** IllegalArgumentException("Input cannot be null");

}

**if** (input.trim().isEmpty()) {

**throw** **new** IllegalArgumentException("Input cannot be empty");

}

**try** {

**int** number = Integer.*parseInt*(input);

**if** (number == 0) {

**throw** **new** ArithmeticException("Cannot divide by zero");

}

// Simulate some operation that could fail

**int** result = 100 / number;

System.***out***.println("Result: " + result);

} **catch** (NumberFormatException e) {

**throw** **new** NumberFormatException("Invalid number format: " + input);

}

}

/\*\*

\* Method that validates age and throws exception for invalid values

\* **@param** age the age to validate

\* **@throws** IllegalArgumentException if age is negative or unrealistic

\*/

**public** **void** validateAge(**int** age) {

**if** (age < 0) {

**throw** **new** IllegalArgumentException("Age cannot be negative");

}

**if** (age > 150) {

**throw** **new** IllegalArgumentException("Age cannot exceed 150 years");

}

}

/\*\*

\* Method that processes an array and throws exception for null or empty arrays

\* **@param** array the array to process

\* **@throws** NullPointerException if array is null

\* **@throws** IllegalArgumentException if array is empty

\*/

**public** **void** processArray(**int**[] array) {

**if** (array == **null**) {

**throw** **new** NullPointerException("Array cannot be null");

}

**if** (array.length == 0) {

**throw** **new** IllegalArgumentException("Array cannot be empty");

}

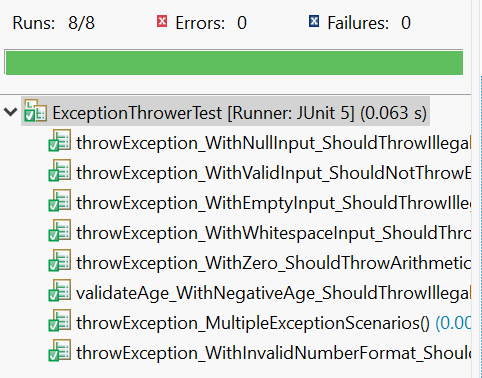
// Process the array (dummy implementation)

System.***out***.println("Processing array with " + array.length + " elements");

}

}

Output :



Exercise 5: Timeout and Performance Testing

Scenario: You want to ensure that a method completes within a specified time limit.

Code :

**package** PerformanceTester;

//This package contains tests for timeout functionality.

**import** java.time.Duration; // Import for handling time durations.

**import** java.util.concurrent.TimeUnit; // Import for time-related utility classes.

**import** org.junit.jupiter.api.Assertions; // Import for JUnit assertions.

**import** org.junit.jupiter.api.Test; // Import for JUnit test annotations.

**import** org.junit.jupiter.api.Timeout; // Import for setting test timeouts.

//This class demonstrates different ways to apply timeouts in JUnit tests.

**class** PerformanceTester {

//Section 1: Using Timeout() annotation

// Test 1: Demonstrates using Timeout annotation with Thread.sleep

@Test

@Timeout(6) // Set timeout to 6 seconds

**public** **void** test1() **throws** InterruptedException {

Thread.*sleep*(5000); // Simulate a 5-second delay

System.***out***.println("Test1 Passed within the time");

}

// Test 2: Demonstrates using Timeout annotation with TimeUnit.SECONDS.sleep

@Test

@Timeout(6) // Set timeout to 6 seconds

**public** **void** test2() **throws** InterruptedException {

TimeUnit.***SECONDS***.sleep(5); // Another way to simulate a 5-second delay

System.***out***.println("Test2 Passed within the time");

}

//Section 2: Using Assertions

// Test 3: Demonstrates using Assertions.assertTimeout

@Test

**public** **void** test3() {

Assertions.*assertTimeout*(Duration.*ofSeconds*(7), () -> delaySeconds(6)); // Assert a timeout of 7 seconds for a

// 6-second delay

System.***out***.println("Test3 Passed within the time");

}

// Helper method to simulate delays

**private** **void** delaySeconds(**int** seconds) **throws** InterruptedException {

TimeUnit.***SECONDS***.sleep(seconds);

}

//Section 3: Demonstrating timeout failures

//Test 4: Using Assertions.assertTimeout to handle timeout failure

@Test

**public** **void** test4() {

Assertions.*assertTimeout*(Duration.*ofSeconds*(4), () -> {

**try** {

Thread.*sleep*(5000); // Delay of 5 seconds, expected to fail

} **catch** (InterruptedException e) {

// Handle interruption if needed

}

});

}

//Test 5: Same as Test 4, using TimeUnit.SECONDS.sleep

@Test

**public** **void** test5() {

Assertions.*assertTimeout*(Duration.*ofSeconds*(4), () -> {

**try** {

TimeUnit.***SECONDS***.sleep(5); // Delay of 5 seconds, expected to fail

} **catch** (InterruptedException e) {

// Handle interruption if needed

}

});

}

// Test 6: Using Assertions.assertTimeout with a shorter timeout than delay

@Test

**public** **void** test6() {

Assertions.*assertTimeout*(Duration.*ofSeconds*(5), () -> delaySeconds1(6));

// Assert a timeout of 5 seconds for a 6-second delay

}

// Helper method for Test 6

**private** **void** delaySeconds1(**int** seconds) **throws** InterruptedException {

TimeUnit.***SECONDS***.sleep(seconds);

}

}

Output :

