Java JUnit

Unit Testing

Unit testing is a critical part of software development, ensuring that individual components of the software work as expected. In Java, unit testing can be done using various frameworks, with JUnit being one of the most popular.

JUnit

- For setting up JUnit 5, add the Maven dependency junit-jupiter-api to your pom.xml file
- JUnit 5 provides annotations and assertions to help you write tests. Here are some basic annotations:
  - @Test: Marks a method as a test method.
  - @BeforeEach: Runs before each test method.
  - @AfterEach: Runs after each test method.
  - @BeforeAll: Runs before all test methods in the class (must be static).
  - @AfterAll: Runs after all test methods in the class (must be static).
- Simple example:

Here's a simple example of a class to be tested:

java Copy code

```java
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public int subtract(int a, int b) {
        return a - b;
    }
}
```

And here is a corresponding test class:

java Copy code

```java
import org.junit.jupiter.api.*;

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

class CalculatorTest {

    private Calculator calculator;

    @BeforeEach
    void setUp() {
        calculator = new Calculator();
    }

    @Test
    void testAdd() {
        assertEquals(5, calculator.add(2, 3));
    }

    @Test
    void testSubtract() {
        assertEquals(1, calculator.subtract(3, 2));
    }
}
```

- JUnit provides a variety of assertions to check the expected results:
  - assertEquals(expected, actual): Asserts that two values are equal.
  - assertNotEquals(unexpected, actual): Asserts that two values are not equal.
  - assertTrue(condition): Asserts that a condition is true.
  - assertFalse(condition): Asserts that a condition is false.
  - assertNull(object): Asserts that an object is null.
  - assertNotNull(object): Asserts that an object is not null.
  - assertThrows(expectedTypeException, lambdaExecutable): Asserts that an executable throws a specific exception.
- assertThrows example: grade of negative marks scenario, 'var' is just new java 11 feature to replace class name, it uses lambda

```
67      @Test
68      void negativeOneShouldReturnIllegalArgumentExcept
69          var grader = new Grader();
70          assertThrows(IllegalArgumentException.class,
71                  () -> {
72                      grader.determineLetterGrade(-1);
73                  });
```

- JUnit 5 supports parameterized tests, allowing you to run a test multiple times with different parameters:

```java
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

class ParameterizedTestExample {

    @ParameterizedTest
    @ValueSource(strings = {"racecar", "radar", "able was I ere I saw elba"})
    void testPalindrome(String candidate) {
        assertTrue(isPalindrome(candidate));
    }

    boolean isPalindrome(String str) {
        String reversed = new StringBuilder(str).reverse().toString();
        return str.equals(reversed);
    }
}
```

- Basic annotations of JUnit controlling lifecycle:

Console | TestCalculator.java | Calculator.java | TestSciCalculator.java

```java
    @Autowired
    private Calculator calc;

    @BeforeAll
    static void initAll() {
        System.out.println("Before all total tests");
    }

    @BeforeEach
    void init() {
        System.out.println("Before each individual test");
    }

    @AfterEach
    void tearDown() {
        System.out.println("After each individual test");
    }

    @AfterAll
    static void tearDownAll() {
        System.out.println("After all total tests");
    }

    @Test
    public void shouldAdd2Nums() {
        int x = 4, y = 15;
```

Springboot Application JUnit testing using Mockito Framework

- Mockito allows you to mock/imitate external dependencies like database and test your controllers, services, and repositories in isolation
- Steps are:

EmployeeStream.java | StreamProgramsNumber... | ChelseaAppApplication.ja... | ChelseaController.java | ChelseaAppApplicationTe... | ChelseaServiceImpl.java | ChelseaService.java | Chelsea.java

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
class ChelseaAppApplicationTests {

    @MockBean
    private ChelseaRepo mockChelseaRepo; //mocking repo

    @Autowired //can be @InjectMocks too in some cases
    private ChelseaService chelseaService; //testing service

    @Test
    public void testGetPlayerByKitId() throws PlayerNotFoundException { //just testing 1 method of interface chelsea se

        //creating db-expected data of 'Chelsea'
        Chelsea che = new Chelsea();
        che.setId(8); che.setName("Lampardss"); che.setRating(9.8);

        //using mockito to say that whenever our repo hits that jpa method 'findById(8)',
        //return our db-expected, instead of actually hitting DB
        when(mockChelseaRepo.findById(8)).thenReturn(Optional.of(che));

        //testing our actual service class chelseaService's method
        Chelsea actualResult = chelseaService.getPlayerByKitId(8);

        assertTrue(null!=actualResult);//if service returned <Optional>,then assertTrue(actualResult.isPresent())

        //Finally, testing assertEquals(expected,actual) [in reality: assertEquals(expected,db-expected)
        //where expected= our custom input, actual= db-expected output by service which depends on mock repo now
        assertEquals("Lampardss", actualResult.getName()); //if it really hit db, actual output would be 'Lampard'

```