TDD:
JUnit Introduction

"Code with Passion!"

Topics

- What is JUnit?
- JUnit 5 annotations
  - @Test
  - @BeforeEach, @AfterEach
  - @BeforeAll, @AfterAll
  - @Disabled, @DisplayName
- Assert statements
- AssertThat
- Testing tools
- Exception handling
- Parameterization
- Best practices

What is JUnit?

What is JUnit?

- Unit testing framework for Java
- De-facto standard for unit testing
- Free and open-sourced

JUnit 5 Annotations

@Test

@BeforeEach, @AfterEach

@BeforeAll, @AfterAll

@Disabled

How to write JUnit test?

- Create test class with test methods
- Each test method has method signature
  - Annotate it with *@Test*
  - public
  - void return type
  - No arguments
- Add fixtures
  - *@BeforeEach* and *@AfterEach* to run before/after each test method
  - *@BeforeAll* and *@AfterAll* to run once before/after all test methods
  - *@Disabled* to temporarily ignore the testing

Example: Calculator Test Target

```
public class Calculator {

    public int add(int x, int y) {
        return x + y;
    }

    public int subtract(int x, int y) {
        return x - y;
    }
}
```

Example: Test class of Calculator Test Target

```
import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CalculatorTest {
    private Calculator calculator;

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

    @Test
    void should_return_added_value_when_two_numbers_are_given() {
        assertEquals(5, calculator.add(2, 3));                          // using Junit 5 Jupiter  Assertions package
        Assertions.assertThat(calculator1.add(2, 3)).isEqualTo(5); // using AssertJ Assertions package
    }

    @Test
    void should_return_subtracted_value_when_two_numbers_are_given() {
         assertEquals(-1, calculator.subtract(2, 3));                          // using Junit 5 Jupiter Assertions package
        Assertions.assertThat(calculator1.subtract(2, 3)).isEqualTo(-1); // using AssertJ Assertions package
    }
}
```

Assert Statements

Assert Statements from Junit 5 Jupiter

- JUnit Assertions are methods starting with *assert*
- Determines the success or failure of a test
- An assert is simply a comparison between an expected value and an actual value
- Two variants
  - assertXXX(...)
  - assertXXX(…, String message) - the message is displayed when the assertXXX() fails
  - assertXXX(…, <Lambda-expression)

Assert Statements from Junit 5 Jupiter Package

- Asserts expected.equals(actual) behavior
  - assertEquals(expected, actual)
  - assertEquals(expected, actual, String message)
- Asserts expected == actual behavior
  - assertSame(Object expected, Object actual)
  - assertSame(Object expected, Object actual, String message,)
- Asserts that a condition is true
  - assertTrue(boolean condition)
  - assertTrue(boolean condition, String message)
- Asserts that a condition is false
  - assertFalse(boolean condition)
  - assertFalse(boolean condition, String message)

Assert Statements from Junit 5 Jupiter Package

- Asserts object reference is null
  - assertNull(Object obj)
  - assertNull(Object obj, String message)
- Asserts object reference is not null
  - assertNotNull(Object obj)
  - assertNotNull(Object obj, String message)

AssertJ Assertion Package

What is and Why AssertJ?

- Open-source assertion package for Java
- Fluent and rich assertions in Java tests
- Much more readable than other assertions packages
- Default assertion package in “spring-boot-starter-test”

Example: AssertJ Examples

- assertThat(result).isEqualTo(5);
- assertThat(resultObject.getName()).startsWith(“Yo”).endsWith(“Ya”);
- assertThat(resultObject).isEqualTo(expectedObject);
- assertThat(resultBoolean).isTrue();
- assertThat(resultList).contains(“John”);
- assertThat(resultList).isNotEmpty();
- assertThat(resultList).isNotEmpty().contains(“John”).doesNotContain(“Tom”);
- assertThat(resultMap).isNotEmpty().containsKey(3);

Testing Tools

Testing Tools

- Automatic testing tools
  - Each time a change is made on the source code, tests are run
  - Infinitest
- Test generating tools
  - MoreUnit
- Code coverage tools
  - Used to describe the degree to which the source code of a program is executed when a particular test suite runs
  - EclEmma, JaCoco
- Source code version control
  - Git
- Continuous Build tools
  - Jenkins

Lab:

Exercise 2: Installation of Tools 1651_tdd_junit.zip

Exception Handling

Testing Exception in JUnit 5

```
@Test void testThrowsException() throws Exception {

      Assertions.assertThrows(Exception.class,
                                          () -> {
                                                  // Lambda expression
                                           });

}
```

Lab:

Exercise 3: Exception Handling
1651_tdd_junit.zip

Parameterization

Parameterized Testing

- The Parameterized Test is used when we find ourselves writing identical tests where only a few data input values are different but the logic is the same
  - Only one test method is needed which will have parameterized data supplied to it

- How to do parameterized testing
  - Annotate the test method with @ParameterizedTest

Example: Parameterized testing

```
public class CalculatorTest {

    @DisplayName("parameterized test")
    @ParameterizedTest(name = "{0} + {1} = {2}")
    @CsvSource({
            "0,    1,   1",
            "1,    2,   3",
            "49,  51, 100",
            "1,  100, 101"
    })
    void add(int first, int second, int expectedResult) {
          Calculator2 calculator = new Calculator2();
          assertEquals(expectedResult, calculator.add(first, second),
                              () -> first + " + " + second + " should equal " + expectedResult);
    }
}
```

Use this annotation

Lab:

Exercise 4: Parameterization
1651_tdd_junit.zip

JUnit Best Practices

Test Code vs Production Code

- Your test code becomes part of the codebase
- It is backed up and put into a source code repository along with the production code (test targets)
- Test code needs to be documented, although not as heavily as the production code
- Note that test code will most likely not deploy with the production code

Be thorough

- Write tests for everything
  - Except methods that are "too simple to fail"
  - Only trivial getter/setter methods are too simple to fail

- Write multiple tests for each method
  - "positive" testing
  - "negative" testing
  - boundary value testing (BVT)

Code with Passion!