Introduction to Unit Testing and JUnit5

Affan Rauf

December 18, 2024

What is Unit Testing?

- Unit testing is a software testing technique that tests individual units of code (e.g., methods or functions) in isolation.
- A "unit" is the smallest testable part of an application, usually a function or a method.
- Unit tests help to ensure that code works as expected, is maintainable, and reduces bugs.

Why Unit Testing is Important?

- Ensures that each unit of the code works correctly.
- Facilitates early detection of bugs.
- Simplifies refactoring by ensuring existing functionality remains intact.
- Serves as documentation for how units of code behave.
- Forms the foundation for Test-Driven Development (TDD).

What is JUnit5?

- JUnit5 is a popular testing framework for Java, designed to test Java applications.
- It includes support for Java 8 features.

Basic Structure of a JUnit5 Test Class

- `@Test` - Marks a method as a test case.
- `@BeforeEach` - Method executed before each test.
- `@AfterEach` - Method executed after each test.
- `@BeforeAll` - Static method executed once before all tests.
- `@AfterAll` - Static method executed once after all tests.

Example of a JUnit5 Test

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

class CalculatorTest {

    @Test
    void testAddition() {
        Calculator calc = new Calculator();
        int result = calc.add(2, 3);
        assertEquals(5, result);
    }
}
```

Assertions in JUnit5

- `assertEquals(expected, actual)` - Asserts that two values are equal.
- `assertTrue(condition)` - Asserts that the condition is true.
- `assertFalse(condition)` - Asserts that the condition is false.
- `assertThrows(exception.class, () -> method)` - Asserts that a specific exception is thrown.

@BeforeAll and @AfterAll Overview

- In JUnit5, methods annotated with `@BeforeAll` and `@AfterAll` are executed once before and after all test methods in the test class, respectively.
- These methods are used to set up or clean up resources that are shared across multiple test cases.
- Unlike `@BeforeEach` and `@AfterEach`, these methods do not run before or after every single test.

Why @BeforeAll and @AfterAll Must Be Static

- JUnit5 creates a new instance of the test class for each test method.
- Since `@BeforeAll` and `@AfterAll` are designed to run once for all tests, they need to be independent of individual test instances.
- Static methods belong to the class itself, not to any specific instance, making them suitable for tasks that are shared across the entire test suite.

Example Use Case for Static @BeforeAll

- Example: Connecting to a shared database once for all tests in a class.
- This prevents the overhead of reconnecting before each individual test.

```
@BeforeAll
static void setUpDatabaseConnection() {
    Database.connect();
}
```

Example Use Case for Static @AfterAll

- Example: Closing a database connection or releasing resources after all tests.
- This ensures the resources are properly cleaned up after all test methods have executed.

```
@AfterAll
static void tearDownDatabaseConnection() {
    Database.disconnect();
}
```

Key Points

- `@BeforeAll` and `@AfterAll` should be used for actions that are performed only once, not before/after each test.
- These methods must be static to avoid reliance on test class instances.
- They are ideal for tasks like establishing or releasing global resources (e.g., database connections, opening files).

Overview of @BeforeEach and @AfterEach

- `@BeforeEach` - Executes before each test method in the test class.
- `@AfterEach` - Executes after each test method in the test class.
- Both annotations are used to set up and tear down resources or configurations for individual test cases.

Common Tasks in @BeforeEach

- Setting up resources required by each test method.
- Initializing variables, objects, or states before each test.

Example of @BeforeEach

- Example: Resetting an object or data structure before every test.

```
@BeforeEach
void setUp() {
    calculator = new Calculator();
    list = new ArrayList<>();
}
```

- In this example, a new `Calculator` object and an empty `ArrayList` are created before each test.

Common Tasks in @AfterEach

- Cleaning up or resetting states or resources used by the test.
- Removing or nullifying references to objects or data to prevent side effects.

Example of @AfterEach

- Example: Clearing or resetting data after each test.

```
@AfterEach
void tearDown() {
    list.clear();  // Clear the list after each test
}
```

- In this example, the `ArrayList` is cleared after each test to ensure no data remains for the next test.

Key Points

- `@BeforeEach` is used to prepare or set up everything needed for each individual test case.
- `@AfterEach` is used to clean up, reset, or release resources after each test.
- These methods are essential to ensure test isolation and avoid side effects between tests.

Introduction to Unit Test Properties

- A good unit test should verify the correctness of a small unit of code.
- Unit test properties can be categorized into:
  1. Properties related to choosing the right test values.
  2. Properties related to the proper execution of the tests.

Choosing the Right Test Values

- Correct test values are crucial for ensuring the validity of the unit test.
- These properties ensure that the test is comprehensive and meaningful.

Properties of Choosing the Right Test Values

- **Coverage of Different Scenarios:**
  - Test a wide range of input scenarios, including:
    - Normal or expected values.
    - Edge cases (e.g., minimum, maximum inputs).
    - Exceptional cases (e.g., null, invalid inputs).
- **Representativity:**
  - Ensure selected test cases represent different conditions the function might face in production.
- **Boundary Testing:**
  - Test at the boundaries of valid input ranges to identify potential off-by-one or limit issues.

Proper Execution of Unit Tests

- Proper execution ensures that the test delivers reliable and reproducible results.
- This category focuses on execution best practices for running tests efficiently and in isolation.

Properties of Proper Execution

- **Test Isolation:**
  - Tests should be independent, so one test's success or failure does not affect others.
  - Avoid shared state between tests unless explicitly reset.
- **Repeatability:**
  - Tests should produce the same results when run multiple times.
  - Avoid dependence on external factors like system time, network, or database states.
- **Fast Execution:**
  - Unit tests should be fast to execute to ensure that large test suites can run frequently.
  - Long-running tests should be avoided or separated from unit tests.
- **Clear Assertion:**
  - Each test should contain clear assertions that confirm expected behavior.
  - Tests should fail only when the code is incorrect, not due to external factors or ambiguous conditions.

Key Takeaways

- Choosing the right test values ensures thorough coverage and realistic scenarios.
- Proper execution ensures test reliability, fast feedback, and independence between tests.
- A combination of both categories leads to high-quality, maintainable unit tests.