JUnit & Unit Testing in Java

What is JUnit?

JUnit is a widely-used **testing framework** in Java, particularly designed for writing and running unit tests. Here's what it offers:

- **Assertion methods**: These methods are used to verify the expected output of your code. Some common methods include `assertTrue()`, `assertFalse()`, and `assertEquals()`.
- **Annotations**: JUnit provides annotations like `@Test`, which marks methods as tests.
- **Automated Testing**: Once you write tests, JUnit automates the process of running these tests.

JUnit4 vs JUnit5

- **JUnit4**: Released in 2006, it was a single monolithic system designed for older versions of Java (Java SE 6).
- **JUnit5**: Released in 2017, JUnit5 is modular, more flexible, and supports newer versions of Java (Java SE9+). However, JUnit5 is **not backward compatible** with JUnit4.

⇒ Prefer **Junit5** whenever possible.

Setting up JUnit in Maven

To use JUnit in your project, you'll need to add JUnit as a dependency in your project's **Maven POM** file:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.x.x</version>
    <scope>test</scope>
</dependency>
```

After updating the POM file, right-click on `pom.xml` and reload the Maven project.

Basic JUnit Usage

JUnit tests are created by adding **test methods** in separate test classes. Here's the structure:

- **@Test annotation**: Marks a method as a test.
- **Assertion methods**: Used within the test to verify the correctness of the code.

Example:

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

public class MyTest {
    @Test
    public void testSomething() {
        assertTrue(true);
    }
}
```

Here, the `assertTrue(true)` method checks if the condition is `true`. If it's not, the test will fail.

Writing Unit Tests

The process of writing a unit test generally follows three steps:

1. **Create an instance of the class you're testing**.
2. **Call methods on the instance** to change its state or obtain a result.
3. **Use assertions** to verify the behavior of the methods.

For example:

```
@Test
public void testAddValue() {
    PrimeCounter pc = new PrimeCounterImpl();
```

```
    pc.addValue(11);
    assertTrue(pc.isPrime()); // Checks if 11 is prime
}
```

This test verifies that after adding 11, the `isPrime()` method should return `true`.

Assertions in JUnit

JUnit provides various assertion methods to test different conditions:

- `assertTrue(condition)`: Throws an exception if the condition is `false`.
- `assertEquals(expected, actual)`: Throws an exception if `actual` is not equal to `expected`.
- `assertNull(object)`: Throws an exception if the object is not `null`.
- `assertNotNull(object)`: Throws an exception if the object is `null`.

Here's an example:

```
@Test
public void testEquality() {
    assertEquals(5, 5); // Test passes
    assertEquals("hello", "world"); // Test fails
}
```

Handling Exceptions in JUnit

In JUnit, there are situations where you want to **test if a method throws an exception** when it's supposed to. This is a common use case when you want to ensure your program properly handles invalid input or other error conditions.

1. Using `@Test(expected = Exception.class)` (JUnit4):

- In JUnit4, you can use the `expected` attribute of the `@Test` annotation to specify the exception that should be thrown by the method under test.
- Here's an example where we expect an `IllegalArgumentException`:

```
@Test(expected = IllegalArgumentException.class)
public void testMethodThrowsException() {
    MyClass obj = new MyClass();
    obj.someMethod(-1); // Passing invalid input that shou
ld throw an exception
}
```

In this test:

- The method `someMethod(-1)` is expected to throw an `IllegalArgumentException`.
- If it does, the test passes.
- If no exception or a different exception is thrown, the test fails.

2. Using `assertThrows` (JUnit5):

- JUnit5 introduced a better and more flexible way to handle exceptions using the `assertThrows()` method. This allows you to not only verify that an exception was thrown, but also lets you inspect the exception message or other properties of the exception.
- Here's an example:

```
@Test
public void testMethodThrowsException() {
    MyClass obj = new MyClass();

    // This method should throw an IllegalArgumentExceptio
n
    Exception exception = assertThrows(IllegalArgumentExce
ption.class, () -> {
        obj.someMethod(-1);
    });

    // Optionally, you can also assert that the exception
message matches your expectations
```

```
    assertEquals("Invalid input", exception.getMessage());
}
```

Explanation:

- `assertThrows()` takes two arguments:
  - The type of exception you expect (e.g., `IllegalArgumentException.class`).
  - A lambda expression or method reference that runs the code under test.
- It returns the actual exception that was thrown, which you can then inspect further (e.g., to check the message or other properties).

3. Example of Testing Exceptions with a Custom Class:

- Let's say we have a `BankAccount` class with a method `withdraw()` that throws an `InsufficientFundsException` if the account balance is too low.

```
public class BankAccount {
    private double balance;

    public BankAccount(double balance) {
        this.balance = balance;
    }

    public void withdraw(double amount) {
        if (amount > balance) {
            throw new InsufficientFundsException("Insuffic
ient funds for withdrawal");
        }
        balance -= amount;
    }
}
```

- We can write a JUnit5 test to verify that the correct exception is thrown:

```
@Test
public void testWithdrawInsufficientFunds() {
    BankAccount account = new BankAccount(100);

    // Verify that InsufficientFundsException is thrown
    InsufficientFundsException exception = assertThrows(In
sufficientFundsException.class, () -> {
        account.withdraw(150); // Trying to withdraw more
than the balance
    });

    // Check the exception message
    assertEquals("Insufficient funds for withdrawal", exce
ption.getMessage());
}
```

In this test:

- We attempt to withdraw more money than is available in the account.
- The `assertThrows` method ensures the `InsufficientFundsException` is thrown and verifies the exception message.

Test Coverage

When writing tests, you want to achieve **high test coverage**, meaning you write enough tests to cover as many situations (expected and edge cases) as possible. For example:

- **Expected cases**: Normal behavior of the methods.
- **Edge cases**: Extreme or unusual situations, like passing negative numbers, zero, or very large values.

Unit Testing Example: PrimeCounter

Suppose you need to implement a `PrimeCounter` class. Before writing the class, we can first write tests based on the interface:

```
public interface PrimeCounter {
    void addValue(int value);
    boolean isPrime();
}
```

Test for constructor and initial state:

```
@Test
public void PrimeCounterImplTest01() {
    PrimeCounter pc = new PrimeCounterImpl();
    assertFalse(pc.isPrime()); // Initially, the counter shou
ld not be prime
}
```

Test for adding a value:

```
@Test
public void PrimeCounterImplTest02() {
    PrimeCounter pc = new PrimeCounterImpl();
    pc.addValue(11);
    assertTrue(pc.isPrime()); // After adding 11, it should b
e prime
}
```

Test for handling negative values:

```
@Test
public void PrimeCounterImplTest03() {
    PrimeCounter pc = new PrimeCounterImpl();
    pc.addValue(-3);
    assertFalse(pc.isPrime()); // Negative numbers should not
be prime
}
```

Unit Testing in Isolation

Each unit test should ideally test **one specific aspect** of a class. For example, test a method independently, without relying on the results of other methods. However, it’s often okay to call multiple methods within a single test if that makes sense for testing.