Appendix

Contents

Appendix

1 Course on Unit Testing and JUnit

1.1 Introduction to Unit Testing

Unit testing is an essential practice in software development. It helps verify that a unit of code (typically a function or method) works as expected. In this course, we will explore:

- The definition and importance of unit tests.
- The JUnit framework for testing in Java.
- Handling scenarios where information is missing or needs to be added.

1.2 What is a Unit Test?

A unit test is an automated test that validates the behavior of a specific code unit. A unit is often defined as:

- A method in a class.
- A function in a program.

1.2.1 Why Perform Unit Tests?

Unit tests offer several advantages:

- **Early bug detection:** Errors are identified before the code is integrated with other modules.
- **Ease of maintenance:** When code is modified, tests ensure that no new errors are introduced.
- **Code documentation:** Tests serve as a reference to understand how each method is supposed to behave.

1.3 Overview of JUnit

JUnit is a widely used Java testing framework for writing and running unit tests. It is lightweight, fast, and easily integrates with modern tools.

1.3.1 Key JUnit Annotations

Here are the main annotations used in JUnit:

- `@Test`: Marks a method as a unit test.
- `@BeforeEach`: Method executed before each test, useful for initializing data.
- `@AfterEach`: Method executed after each test, often used for cleaning up resources.
- `@BeforeAll`: Method executed once before all tests.

- `@AfterAll`: Method executed once after all tests.
- `@Disabled`: Temporarily disables a test.

1.3.2 Main Assertions

Assertions are methods that check the test results:

- `assertEquals(expected, actual)`: Checks that the actual value matches the expected value.
- `assertTrue(condition)`: Checks that a condition is true.
- `assertFalse(condition)`: Checks that a condition is false.
- `assertThrows(exception.class, () -> {...})`: Verifies that a specific exception is thrown.

1.4 Example Using JUnit

Here is a simple example testing an addition method in a calculator.

1.4.1 Calculator Class Code

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

1.4.2 Unit Test Code

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

class CalculatorTest {
    @Test
    void testAddition() {
        Calculator calculator = new Calculator();
        int result = calculator.add(2, 3);
        assertEquals(5, result); // Verify that 2 + 3 = 5
    }
}
```

2 Mock Objects with Mockito

2.1 What is a Mock Object?

A **mock object** is a **simulation** of an external dependency used in unit tests. It allows simulating the behavior of this dependency without actually executing its associated code. Mocks enable controlling dependency behavior and isolating the code under test.

2.1.1 Why Use Mock Objects?

Mocks are primarily used for:

- **Isolating tests:** When a class or method depends on other external resources (like a database or an API), it is challenging to test with these real dependencies. Mocks simulate these resources and test only the class logic.
- **Controlling dependency outputs:** Mocks can be programmed to return specific values or exceptions to test various scenarios.
- **Avoiding side effects:** Dependencies may have undesirable side effects, like sending emails or modifying a real database. Mocks prevent these during tests.

2.2 Introduction to Mockito

Mockito is a popular Java framework for creating **mock objects** in unit tests. It simplifies the process of simulating dependencies of a class or method. Mockito makes it easy to simulate complex behaviors and verify method calls.

2.2.1 Main Features of Mockito

Mockito offers several essential features for working with mocks:

- **Creating mocks:** Easily create mocks for class dependencies.
- **Stubbing:** Define what mocks should return when a method is called.
- **Verification:** Check if specific methods were called on mocks, useful for testing interactions.

2.3 Basic Examples with Mockito

Creating a Mock:

```
import static org.mockito.Mockito.*;

public class UserServiceTest {
    @Test
    public void testGetUser() {
        UserRepository userRepository = mock(UserRepository.class);
        User mockUser = new User("John", "Doe");
        when(userRepository.findById(1)).thenReturn(mockUser);

        UserService userService = new UserService(userRepository);
        User user = userService.getUser(1);
```

```
        assertEquals("John", user.getFirstName());
    }
}
```

Conclusion

In conclusion, unit tests and mock objects play a central role in modern software development. Using frameworks such as JUnit and Mockito, developers can:

- Ensure code reliability and quality.
- Quickly identify and fix errors in individual units.
- Effectively simulate external dependencies to isolate tests.

Unit tests encourage a proactive approach to bug management and provide clear documentation of functionalities. With mock objects, it becomes possible to test complex scenarios while minimizing risks related to external dependencies. These tools are indispensable for building robust, maintainable, and scalable software.