Unit testing is a fundamental practice in software development that involves testing individual components or units of code to ensure they work as expected. In Java, unit testing is often done using a testing framework like JUnit. I'll explain the basics of Java unit testing with examples.

Basics of Java Unit Testing:

1. **Unit**: A unit in unit testing refers to the smallest testable part of your code. It could be a method, a class, or even a single function.

2. **Testing Framework**: In Java, JUnit is a popular testing framework used for writing and running unit tests. JUnit provides annotations, assertions, and test runners to make testing easy.

3. **Test Case**: A test case is a set of conditions and inputs used to verify that a particular unit of code behaves as expected.

4. **Test Suite**: A test suite is a collection of test cases that can be run together. It allows you to group related tests.

Example 1: Testing a Simple Calculator Class

Let's say you have a simple `Calculator` class with two methods, `add` and `subtract`, and you want to write unit tests for it using JUnit.

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

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

Now, let's write unit tests for this class using JUnit:

```java
// CalculatorTest.java
import org.junit.Test;
import static org.junit.Assert.*;
```

```
public class CalculatorTest {
    @Test
    public void testAdd() {
        Calculator calculator = new Calculator();
        int result = calculator.add(2, 3);
        assertEquals(5, result);
    }

    @Test
    public void testSubtract() {
        Calculator calculator = new Calculator();
        int result = calculator.subtract(5, 3);
        assertEquals(2, result);
    }
}
```

In the above example:

- We import JUnit annotations and assertions.
- We write test methods and annotate them with `@Test`.
- Inside the test methods, we create an instance of the `Calculator` class, call its methods, and use JUnit assertions like `assertEquals` to check if the actual result matches the expected result.

To run these tests, you'd typically use an IDE or a build tool like Maven or Gradle that integrates with JUnit.

Example 2: Testing a User Authentication Service

Let's consider a more complex scenario where you want to test a user authentication service:

```java
// AuthService.java
public class AuthService {
    public boolean authenticateUser(String username, String password) {
        // Code to authenticate the user
        return true; // Placeholder for simplicity
    }
}
```

Now, let's write unit tests for the `AuthService` class:

```java
// AuthServiceTest.java
import org.junit.Test;
import static org.junit.Assert.*;

public class AuthServiceTest {
    @Test
    public void testAuthenticateUserValid() {
        AuthService authService = new AuthService();
        assertTrue(authService.authenticateUser("validUser", "validPassword"));
    }

    @Test
    public void testAuthenticateUserInvalid() {
        AuthService authService = new AuthService();
        assertFalse(authService.authenticateUser("invalidUser", "invalidPassword"));
    }
}
```

In this example, we have two test methods for the `AuthService` class. We use `assertTrue` and `assertFalse` to check if the authentication is working correctly for valid and invalid inputs.

Remember that good unit tests cover various scenarios, including edge cases and error handling. Additionally, you should mock or stub external dependencies when testing to isolate the unit under test.

Unit testing is essential for ensuring the correctness of your code and helps in catching bugs early in the development process.