**Bước 1: Thêm Junit thư viện vào dự án**

* Nếu sử dụng Maven, hãy thêm phần phụ thuộc sau vào tệp pom.xml:

*<dependency>*

*<groupId>Junit</groupId>*

*<artifactId>Junit</artifactId>*

*<version>4.12</version> <!-- Hoặc phiên bản Junit mới nhất -->*

*<scope>test</scope>*

*</dependency>*

* Nếu sử dụng Gradle, hãy thêm phần phụ thuộc sau vào tệp build.gradle:

*testImplementation 'Junit:Junit:4.12' // Hoặc phiên bản Junit mới nhất*

**Bước 2: Viết các bài kiểm tra (kiểm tra đơn vị)**

Tạo các bài kiểm tra lớp tương ứng với lớp cần kiểm tra và sử dụng các chú thích của Junit như @Test, @Before, và @Afterđể đánh dấu các phương pháp kiểm tra và thiết lập trước khi chạy bài kiểm tra.

Trong các bài kiểm tra, hãy sử dụng các xác nhận phương thức của Junit như assertEquals, assertTrue, assertFalse để so sánh kết quả thực tế với kết quả mong đợi.

**Bước 3: Chạy bài kiểm tra**

Tùy thuộc vào môi trường phát triển tùy thuộc vào, bạn có thể chạy bài kiểm tra thông qua IDE hoặc dòng lệnh.

Kết quả kiểm tra kết quả sẽ hiển thị thông tin qua các thông báo của Junit, báo cáo về số kiểm tra thành công, số kiểm tra thất bại và chi tiết về từng bài kiểm tra. Các bài kiểm tra được điều chỉnh và ưu tiên tối ưu để đảm bảo kiểm tra hiệu quả và đầy đủ của dự án.

**Những thuật ngữ cần biết khi áp dụng Junit**

Trong Junit, người dùng nên chú ý đến một số thuật ngữ dưới đây để sử dụng framework kiểm tra hiệu quả. Dưới đây là một số khái niệm quan trọng:

* Test Case (Bài kiểm tra): Đây là trường hợp cụ thể mà bạn muốn kiểm tra trong mã nguồn của bạn. Một bài kiểm tra thường tương ứng với một phương thức trong lớp kiểm tra.
* Test Suite (Bộ kiểm tra): Đây là nhóm các bài kiểm tra mà bạn muốn chạy cùng nhau. Bằng cách sử dụng Test Suite, bạn có thể chỉ định các bài kiểm tra cụ thể nào cần phải chạy.
* Annotations (Chú thích): Junit sử dụng các chú thích để đánh dấu các phương thức kiểm tra và thiết lập trước/sau khi chạy kiểm tra.
  + Ví dụ: @Test, @Before, @After, @BeforeClass, và @AfterClass là những annotation quan trọng trong Junit.
* Assertions (Các khả năng so sánh): Junit cung cấp một số phương thức assertions để kiểm tra kết quả của bài kiểm tra.
  + Ví dụ, assertEquals, assertTrue và assertFalse là những phương thức assertions thường được sử dụng.
* Test Runner (Bộ chạy kiểm tra): Đây là công cụ hoặc lớp mà bạn sử dụng để chạy bài kiểm tra của bạn. Junit có các Test Runner mặc định và cũng cho phép tùy chỉnh Test Runner.
* Parameterized Tests (Bài kiểm tra tham số hóa): Đây là một tính năng của Junit cho phép bạn chạy cùng một bài kiểm tra với nhiều bộ tham số khác nhau.
* Mocking and Stubbing (Mock và Stub): Junit thường được kết hợp với các framework khác như Mockito để tạo ra các đối tượng giả lập (mock objects) và đặt giá trị trả về (stubs) trong các bài kiểm tra.
* Fixtures: Thuật ngữ đề cập đến những điều kiện ban đầu cần thiết để thực hiện bài kiểm tra một cách đáng tin cậy. Trong đó bao gồm hoạt động thiết lập dữ liệu ban đầu, tạo ra các đối tượng cần thiết và chuẩn bị môi trường thực thi cho bài kiểm tra.
  + Trong Junit, bạn có thể sử dụng các annotations như @Before để thiết lập fixtures trước khi chạy mỗi bài kiểm tra và @After để dọn dẹp fixtures sau khi bài kiểm tra kết thúc.
* Mock: Giả lập đề cập đến việc tạo các cài đặt đối tượng để thay thế các thực thi đối tượng trong bài kiểm tra. Mô hình đối tượng giúp bạn mô phỏng hoạt động của các đối tượng một cách chính xác và kiểm soát dữ liệu đầu vào và đầu ra của chúng.
  + Việc sử dụng các framework như Mockito trong Junit giúp tạo và quản lý các đối tượng mô phỏng một cách hiệu quả.
* Junit Class: Thuật ngữ thường đề cập đến các lớp (lớp) chứa các bài kiểm tra (trường hợp kiểm tra) và các đồ đạc tương ứng. Mỗi phương thức trong lớp kiểm tra đều được đánh dấu bằng chú thích @Test để xác định rằng đó là một bài kiểm tra.
  + Junit Class cũng có thể chứa các phương thức được sử dụng cho công việc cài đặt và dọn dẹp các cố định bằng cách sử dụng các chú thích @Beforevà @After.

**Cấu trúc một test case**

Các trúc mã mà chúng ta nên tuân thủ trong một test case là cấu trúc AAA. Cấu trúc này gồm 3 thành phần:

* Arrange – Chuẩn bị dữ liệu đầu vào và các điều kiện khác để thực thi test case.
* Act – Thực hiện việc gọi phương thức/hàm với đầu vào đã được chuẩn bị ở Arrange và nhận về kết quả thực tế.
* Assert – So sánh giá trị mong đợi và giá trị thực tế nhận được ở bước Act. Kết quả của test case sẽ là một trong hai trạng thái sau:
* PASS: nếu kết quả mong đợi và kết quả thực tế khớp nhau
* FAIL: nếu kết quả mong đợi khác với kết quả thực tế

### Quy ước đặt tên

#### **Tên lớp chứa mã kiểm thử**

Tên lớp chứa mã kiểm thử thường sử dụng hậu tố “Tests” sau tên lớp được kiểm thử. Ví dụ: tên lớp là StockService thì tên lớp chứa mã kiểm thử sẽ là StockServiceTests.

#### **Tên phương thức kiểm thử (test case)**

Theo nguyên tắc, tên phương thức kiểm thử phải giải thích nhiệm vụ rõ ràng. Có thể tham khảo một số quy ướt đặt tên cho phương thức như sau:

1. Sử dụng từ should.

Ví dụ: favouriteStocksShouldbeSaved, todayPriceShouldBeShowed.

@Test

public void favouriteStocksShouldbeSaved() {}

1. Viết theo mẫu Given[Đầu-Vào]When[Hành-Vi]Then[Kết-Quả-Mong-Đợi]. Ví dụ:

@Test

public void GivenNullUsernameWhenCreateStudentThenShouldThrowException() {}

1. Viết theo mẫu when[hành-vi]\_then[Kết-quả]

@Test

public void whenEnterValidUsernameAndPassword\_thenLoginSuccessfully() {}

Cách viết kiểm thử tốt:

1. Mỗi test case nên là một phương thức độc lập, có thể thực thi mà không phụ thuộc vào bất kỳ test case nào khác.
2. Thứ tự thực hiện của mỗi test case không nên ảnh hưởng đến kết quả thực thi (mặc dù có thể).
3. Khi phát hiện bug trong chương trình, hãy viết ngay kiểm thử cho trường hợp xảy ra bug đó để có thể kiểm tra lại sau này.
4. Tên phương thức kiểm thử phải rõ ràng. Vì vậy không phải do dự nếu tên phương thức quá dài.

Vídụ TestDivisionWhenNumPositiveDenomNegative tốt hơn DivisionTest3.

1. Hãy kiểm thử những trường hợp ném ra ngoại lệ (nếu có). Ví dụ WhenDivisionByZeroShouldThrowException.
2. Hãy kiểm thử các trường hợp negative để làm rõ hình thức phản hồi khi đầu vào là dữ liệu không hợp lệ.

Ví dụ 1:

import com.codegym.Calculator;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

class CalculatorTests {

    private final Calculator calculator = new Calculator();

    @Test

    void shouldReturn2When1Plus1() {

        assertEquals(2, calculator.add(1, 1));

    }

}

Giải thích ví dụ:

* @Test là annotation đánh dấu phương thức shouldReturn2When1Plus1() là một test case. Hãy chú ý rằng tên của phương thức test được viết rất rõ là nên trả về kết quả 2 khi 1 cộng 1.
* Ở phần thân của phương thức chứa một dòng mã kiểm tra kết quả của phương thức add() với đầu vào là hai số có giá trị lần lượt là 1 và 1.
* So sánh giá trị thực tế trả về của phương add() với giá trị mong đợi là 2.
* Sau khi chạy test case này, kết quả sẽ là PASS nếu phương thức add(1, 1)) trả về kết quả đúng bằng 2.

Ví dụ 2:

@Test

public void whenAssertingConditions\_thenVerified() {

assertTrue(10 > 5, "10 lớn hơn 5");

}

Ngược lại, phương thức assertFalse được dùng để kiểm tra kết quả của điều kiện có bằng false hay không.

@Test

public void whenAssertingConditions\_thenVerified() {

assertTrue(5 > 10, "5 không lớn hơn 10");

}

Ví dụ 3:

@Test

public void whenAssertingConditions\_thenVerified() {

float actual = 12 / 3.0001f;

float expected = 4;

assertEquals(expected, actual, 0.001f);

Kết quả của phép chia 12 cho 3.001 thì sẽ là một số thực 3.9998667. Kết quả của ca kiểm thử ví dụ trên là PASSED vì chúng ta đã cho phép mức chênh lệch tối đa là 0.001.

Ví dụ 4:

public void whenAssertingArraysEquality\_thenEqual() {

char[] expected = { 'M', 'a', 'n', 'g' };

char[] actual = "Mang".toCharArray();

assertArrayEquals(expected, actual, "Mảng phải giống nhau");

}

Ví dụ 5:

static void throwAnException() {

throw new IllegalArgumentException("Tham số không hợp lệ");

}

Ví dụ: Thời gian

verify(stockRepository, times(2)).count(); // gọi 2 lần

verify(stockRepository, timeout(10)).count(); // 10 mili giây

@Test

public void tryMockitoMock() {

verify(stockRepository).count();

when(stockRepository.count())

.thenReturn(10);

long stockCount = stockRepository.count();

Assertions.assertEquals(10, stockCount);

verify(stockRepository).count();

}