Section 1: Unit Testing with Junit

1. JUnit Step 1 : Why is Unit Testing Important?

Unit testing là quá trình kiểm thử từng đơn vị nhỏ nhất (unit) của một phần mềm, ví dụ như các hàm, phương thức hoặc lớp trong Java. Đây là một phần quan trọng của quy trình phát triển phần mềm, bao gồm cả phát triển Java, vì nó mang lại nhiều lợi ích quan trọng. Dưới đây là một số lý do vì sao unit testing quan trọng trong Java:

1. **Phát hiện lỗi sớm:** Việc thực hiện Unit Testing giúp phát hiện các lỗi trong mã nguồn sớm hơn, trước khi chúng bị lan rộng sang các phần khác của ứng dụng. Điều này giúp giảm thiểu chi phí và thời gian để sửa chữa các lỗi sau này.
2. **Đảm bảo tính chính xác của mã:** Unit testing giúp đảm bảo rằng mã nguồn của bạn hoạt động đúng như mong đợi. Bằng cách viết các bộ kiểm tra cho từng đơn vị nhỏ, bạn có thể kiểm tra logic và tính chính xác của mã Java. Nếu có bất kỳ lỗi nào, unit testing sẽ giúp bạn xác định và sửa chúng sớm.
3. **Hỗ trợ tái cấu trúc và bảo trì:** Việc phát triển unit test cases đòi hỏi bạn phải tách riêng các đơn vị nhỏ của mã nguồn. Điều này giúp tăng tính rời rạc và tái sử dụng của mã, giúp dễ dàng tái cấu trúc và bảo trì trong tương lai. Nếu bạn thay đổi logic hoặc cải thiện một đơn vị cụ thể, bạn chỉ cần chạy lại unit tests để đảm bảo rằng sự thay đổi không ảnh hưởng đến các phần khác của hệ thống.
4. **Tăng tính tin cậy:** Việc có một bộ unit tests tốt giúp tăng tính tin cậy của mã nguồn. Bạn có thể chạy các bộ kiểm tra mỗi khi thay đổi code hoặc trước khi triển khai, để đảm bảo rằng các chức năng quan trọng không bị hỏng. Điều này giúp giảm thiểu sự cố và lỗi trong quá trình phát triển và triển khai phần mềm.
5. **Hỗ trợ trong quá trình phát triển nhóm:** Unit testing là một phần quan trọng của phương pháp phát triển phần mềm nhóm. Khi nhiều lập trình viên làm việc trên cùng một dự án, unit testing giúp đảm bảo tính tương thích và tính chính xác giữa các đơn vị code được viết bởi các thành viên khác nhau. Nó cũng là một cách để chia sẻ kiến thức và mục tiêu chung về chất lượng mã nguồn.
6. **Tăng hiệu suất phát triển:** Mặc dù việc viết unit tests ban đầu có thể tốn thời gian, nhưng nó có thể giảm thiểu thời gian và công sức trong quá trình phát triển toàn bộ dự án. Với unit tests, bạn có thể nhanh chóng phát hiện và khắc phục lỗi, tránh việc phải dò lỗi và gỡ rối mã nguồn phức tạp sau này. Điều này giúp tăng hiệu suất và sự ổn định trong quá trình phát triển.

Tóm lại, unit testing quan trọng trong Java (cũng như trong bất kỳ ngôn ngữ lập trình nào khác) để đảm bảo tính chính xác, tăng tính tin cậy và hiệu suất phát triển của mã nguồn. Nó cũng giúp hỗ trợ quá trình phát triển nhóm và tái cấu trúc mã.

2. assertEquals

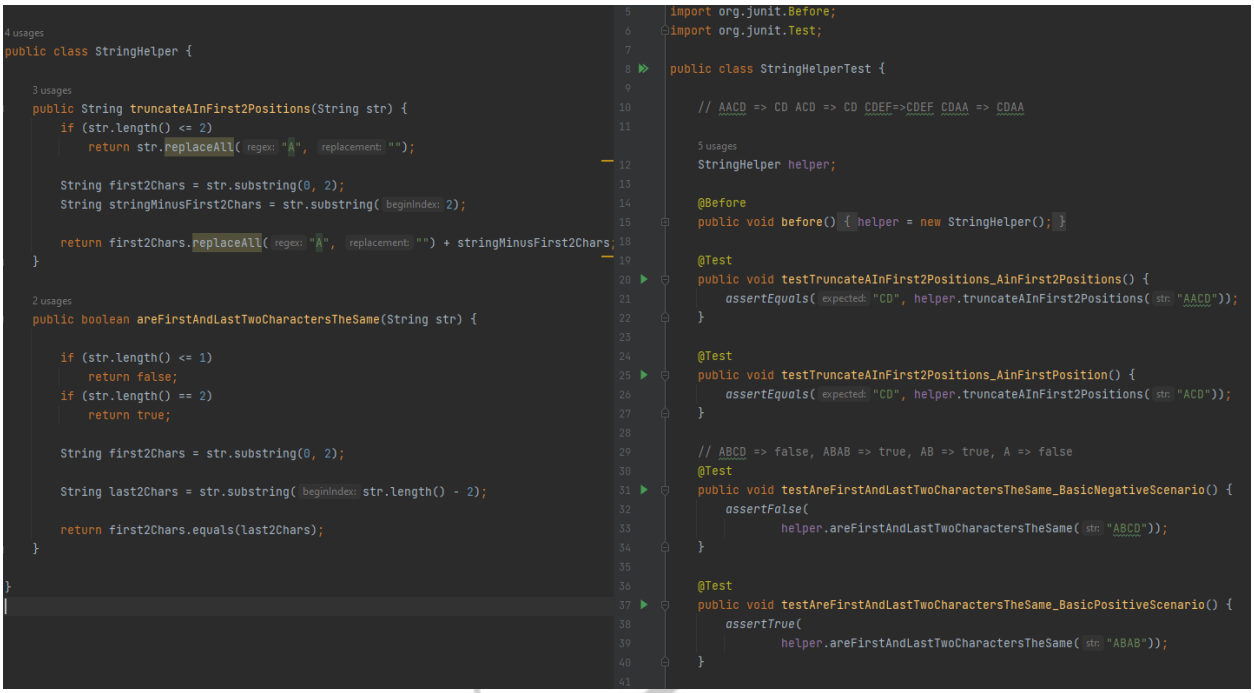
assertEquals là một phương thức trong framework JUnit để so sánh hai giá trị và xác nhận xem chúng có bằng nhau hay không. Phương thức này được sử dụng trong unit testing để kiểm tra tính chính xác của kết quả trả về từ một đơn vị code cụ thể.

Cú pháp của phương thức assertEquals trong JUnit là:

```
assertEquals(expected, actual);
```

- expected là giá trị mà bạn kỳ vọng đúng trong kết quả kiểm tra.
- actual là giá trị thực tế được trả về từ đơn vị code bạn đang kiểm tra.

Phương thức assertEquals sẽ so sánh hai giá trị này với nhau và xác nhận xem chúng có bằng nhau hay không. Nếu chúng bằng nhau, kiểm tra sẽ được coi là thành công và tiếp tục chạy các kiểm tra tiếp theo. Ngược lại, nếu chúng khác nhau, kiểm tra sẽ báo lỗi và hiển thị thông báo rõ ràng về sự sai khác giữa hai giá trị.



```
4 usages
public class StringHelper {

    3 usages
    public String truncateAInFirst2Positions(String str) {
        if (str.length() <= 2)
            return str.replaceAll( regex: "A",  replacement: "");

        String first2Chars = str.substring(0, 2);
        String stringMinusFirst2Chars = str.substring( beginIndex: 2);

        return first2Chars.replaceAll( regex: "A",  replacement: "") + stringMinusFirst2Chars;
    }

    2 usages
    public boolean areFirstAndLastTwoCharactersTheSame(String str) {

        if (str.length() <= 1)
            return false;
        if (str.length() == 2)
            return true;

        String first2Chars = str.substring(0, 2);

        String last2Chars = str.substring( beginIndex: str.length() - 2);

        return first2Chars.equals(last2Chars);
    }

}
```

```
import org.junit.Before;
import org.junit.Test;

public class StringHelperTest {

    // AACD => CD ACD => CD CDEF=>CDEF CDAA => CDAA

    5 usages
    StringHelper helper;

    @Before
    public void before() { helper = new StringHelper(); }

    @Test
    public void testTruncateAInFirst2Positions_AinFirst2Positions() {
        assertEquals( expected: "CD", helper.truncateAInFirst2Positions( str: "AACD"));
    }

    @Test
    public void testTruncateAInFirst2Positions_AinFirstPosition() {
        assertEquals( expected: "CD", helper.truncateAInFirst2Positions( str: "ACD"));
    }

    // ABCD => false, ABAB => true, AB => true, A => false
    @Test
    public void testAreFirstAndLastTwoCharactersTheSame_BasicNegativeScenario() {
        assertFalse(
                helper.areFirstAndLastTwoCharactersTheSame( str: "ABCD"));
    }

    @Test
    public void testAreFirstAndLastTwoCharactersTheSame_BasicPositiveScenario() {
        assertTrue(
                helper.areFirstAndLastTwoCharactersTheSame( str: "ABAB"));
    }

```

3. assertTrue và assertFalse

Trên JUnit, assertTrue và assertFalse là hai phương thức được sử dụng để kiểm tra các điều kiện trong các test case. Dưới đây là mô tả về hai phương thức này:

assertTrue: Phương thức assertTrue kiểm tra xem một biểu thức có đúng (true) hay không. Nếu biểu thức trả về giá trị true, test case được coi là thành công. Ngược lại, nếu biểu thức trả về giá trị false, test case sẽ thất bại.

```
@Test
public void testAreFirstAndLastTwoCharactersTheSame_BasicPositiveScenario() {
    assertTrue(
            helper.areFirstAndLastTwoCharactersTheSame("ABAB"));
}
```

assertFalse: Phương thức assertFalse kiểm tra xem một biểu thức có sai (false) hay không. Nếu biểu thức trả về giá trị false, test case được coi là thành công. Ngược lại, nếu biểu thức trả về giá trị true, test case sẽ thất bại.

```
// ABCD => false, ABAB => true, AB => true, A => false
@Test
public void testAreFirstAndLastTwoCharactersTheSame_BasicNegativeScenario() {
    assertFalse(
          helper.areFirstAndLastTwoCharactersTheSame("ABCD"));
}
```

Trong đó:

```
public boolean areFirstAndLastTwoCharactersTheSame(String str) {

    if (str.length() <= 1)
       return false;
    if (str.length() == 2)
       return true;

    String first2Chars = str.substring(0, 2);

    String last2Chars = str.substring(str.length() - 2);

    return first2Chars.equals(last2Chars);
}
```

3. @Before và @After(@BeforeEach và @AfterEach)

Trong JUnit, @Before và @After là các annotation được sử dụng để đánh dấu các phương thức được thực thi trước và sau mỗi test case. Dưới đây là mô tả về hai annotation này:

@Before: Annotation @Before được sử dụng để đánh dấu một phương thức sẽ được thực thi trước mỗi test case. Phương thức được đánh dấu @Before sẽ chạy trước mỗi test case, và nó được sử dụng để thiết lập trạng thái ban đầu cho các đối tượng và tài nguyên cần thiết cho các test case.

```
public class MyTest {
    private MyClass myObject;

    @Before
    public void setUp() {
       // Khởi tạo đối tượng hoặc thiết lập trạng thái ban đầu
       myObject = new MyClass();
    }

    @Test
    public void testSomething() {
       // Test case
    }
}
```

@After: Annotation @After được sử dụng để đánh dấu một phương thức sẽ được thực thi sau mỗi test case. Phương thức được đánh dấu @After sẽ chạy sau mỗi test case, và nó được sử dụng để giải phóng tài nguyên hoặc thực hiện các công việc sau khi test case hoàn thành.

```
public class MyTest {
    private MyClass myObject;

    @Before
    public void setUp() {
        // Khởi tạo đối tượng hoặc thiết lập trạng thái ban đầu
        myObject = new MyClass();
    }

    @Test
    public void testSomething() {
        // Test case
    }

    @After
    public void tearDown() {
        // Giải phóng tài nguyên hoặc thực hiện các công việc sau khi test
case hoàn thành
        myObject = null;
    }
}
```

4. @BeforeClass và @AfterClass (Trong JUnit 5: @BeforeAll, AfterAll)

Trong JUnit, @BeforeClass và @AfterClass là hai annotation được sử dụng để đánh dấu các phương thức được thực thi trước và sau toàn bộ các test case trong một lớp (class). Dưới đây là mô tả về hai annotation này:

@BeforeClass: Annotation @BeforeClass được sử dụng để đánh dấu một phương thức sẽ được thực thi trước khi tất cả các test case trong lớp bắt đầu chạy. Phương thức được đánh dấu @BeforeClass thường được sử dụng để thiết lập trạng thái ban đầu cho các tài nguyên cần thiết cho toàn bộ lớp.

```
public class MyTest {
    private static MyClass myObject;

    @BeforeClass
    public static void setUpBeforeClass() {
        // Khởi tạo đối tượng hoặc thiết lập trạng thái ban đầu
        myObject = new MyClass();
    }

    @Test
    public void testSomething() {
        // Test case
    }
}
```

Lưu ý rằng phương thức đánh dấu **@BeforeClass** cần được khai báo là static và không có tham số đầu vào.

@AfterClass: Annotation @AfterClass được sử dụng để đánh dấu một phương thức sẽ được thực thi sau khi tất cả các test case trong lớp hoàn thành. Phương thức được đánh dấu @AfterClass thường được sử dụng để giải phóng tài nguyên hoặc thực hiện các công việc sau khi tất cả các test case hoàn thành.

```
public class MyTest {
    private static MyClass myObject;

    @BeforeClass
    public static void setUpBeforeClass() {
        // Khởi tạo đối tượng hoặc thiết lập trạng thái ban đầu
        myObject = new MyClass();
    }

    @Test
    public void testSomething() {
        // Test case
    }

    @AfterClass
    public static void tearDownAfterClass() {
        // Giải phóng tài nguyên hoặc thực hiện các công việc sau khi tất cả
các test case hoàn thành
        myObject = null;
    }
}
```

Lưu ý rằng phương thức đánh dấu **@AfterClass** cần được khai báo là static và không có tham số đầu vào.

5. So sánh Arrays trong JUnit Tests

Trong JUnit, để so sánh hai mảng trong các test case, bạn có thể sử dụng phương thức assertArrayEquals. Phương thức này so sánh các phần tử của hai mảng và xác nhận rằng chúng là bằng nhau. Dưới đây là một ví dụ:

```
public class MyTest {
    @Test
    public void testArrayComparison() {
        int[] expected = {1, 2, 3};
        int[] actual = {1, 2, 3};
        assertArrayEquals(expected, actual);
    }
}
```

Trong ví dụ trên, chúng ta so sánh hai mảng expected và actual bằng cách sử dụng assertArrayEquals(expected, actual). Nếu các mảng có cùng kích thước và các phần tử tương ứng là bằng nhau, test case sẽ được coi là thành công. Ngược lại, nếu có bất kỳ sự khác biệt nào trong các phần tử hoặc kích thước của mảng, test case sẽ thất bại và sẽ hiển thị thông báo lỗi chi tiết.

Lưu ý rằng assertArrayEquals so sánh các phần tử của mảng bằng cách sử dụng phương thức equals. Do đó, nếu mảng chứa các đối tượng không phải là kiểu nguyên thủy, bạn cần đảm bảo rằng đối tượng đó đã triển khai phương thức equals một cách chính xác để cho phép so sánh đúng đắn.

6. Testing Exceptions in JUnit Tests

Trong JUnit, để kiểm tra rằng một ngoại lệ (exception) đã được ném ra trong một test case, bạn có thể sử dụng phương thức **assertThrows**. Phương thức này xác nhận rằng một ngoại lệ đã được ném ra và cho phép bạn kiểm tra loại ngoại lệ và nội dung của nó. Dưới đây là một ví dụ:

```
public class MyTest {
    @Test
    public void testException() {
        // Chuẩn bị
        String input = null;

        // Kiểm tra ngoại lệ
        assertThrows(NullPointerException.class, () -> {
            // Gây ra ngoại lệ
            int length = input.length();
        });
    }
}
```

Trong ví dụ trên, chúng ta sử dụng **assertThrows** để kiểm tra xem một NullPointerException có được ném ra hay không. Một lambda expression được sử dụng để chứa mã gây ra ngoại lệ. Nếu ngoại lệ được ném ra và có cùng loại (NullPointerException), test case sẽ được coi là thành công. Ngược lại, nếu không có ngoại lệ hoặc loại ngoại lệ không phù hợp, test case sẽ thất bại và hiển thị thông báo lỗi chi tiết.

Ngoài ra, bạn cũng có thể sử dụng các phương thức khác như expected trong annotation @Test để kiểm tra ngoại lệ. Dưới đây là một ví dụ:

```
public class MyTest {
    @Test(expected = NullPointerException.class)
    public void testException() {
        // Chuẩn bị
        String input = null;

        // Gây ra ngoại lệ
        int length = input.length();
    }
}
```

Trong ví dụ này, chúng ta sử dụng annotation **@Test** với **expected = NullPointerException.class** để xác định rằng test case này mong đợi một **NullPointerException** được ném ra. Nếu ngoại lệ được ném ra, test case sẽ được coi là thành công. Ngược lại, nếu không có ngoại lệ hoặc ngoại lệ không phù hợp, test case sẽ thất bại.

7. Testing Performance in JUnit Tests

Trong JUnit, để kiểm tra hiệu suất (performance) của một đoạn mã trong test case, bạn có thể sử dụng các phương thức và annotation như Timeout . Dưới đây là một số cách để kiểm tra hiệu suất trong JUnit tests:

Timeout: Bạn có thể sử dụng annotation **@Test** kết hợp với **Timeout** để kiểm tra xem một test case có hoàn thành trong thời gian nhất định hay không. Nếu test case không kết thúc trong khoảng thời gian đã định, nó sẽ được coi là thất bại.

```
import org.junit.Test;

public class MyTest {
    @Test(timeout = 1000) // Giới hạn thời gian 1 giây
    public void testPerformance() {
        // Đoạn mã kiểm tra hiệu suất ở đây
    }
}
```

Trong ví dụ trên, test case **testPerformance** sẽ được thực thi và nếu nó không kết thúc trong 1 giây, nó sẽ được coi là thất bại.

8. Parameterized Tests

Trong JUnit, Parameterized Tests (Test với tham số) cho phép bạn chạy một test case với nhiều bộ dữ liệu khác nhau. Điều này giúp bạn kiểm tra các trường hợp khác nhau của một test case mà không cần viết nhiều mã lặp lại. Dưới đây là cách sử dụng Parameterized Tests trong JUnit:

1. Đánh dấu test class với annotation @RunWith(Parameterized.class).
2. Tạo một phương thức (hoặc một lớp nội) để cung cấp dữ liệu cho các tham số của test case.
3. Tạo một constructor trong test class để nhận các giá trị của các tham số từ phương thức cung cấp dữ liệu.
4. Đánh dấu method test case bằng annotation @Test.
5. Sử dụng các tham số nhận được từ constructor trong test case.

Dưới đây là một ví dụ về cách sử dụng Parameterized Tests trong JUnit:

```
@RunWith(Parameterized.class)
public class StringHelperParameterizedTest {

    // AACD => CD ACD => CD CDEF=>CDEF CDAA => CDAA

    StringHelper helper = new StringHelper();

    private String input;
    private String expectedOutput;

    public StringHelperParameterizedTest(String input, String expectedOutput)
{
       this.input = input;
       this.expectedOutput = expectedOutput;
    }

    @Parameters
    public static Collection<String[]> testConditions() {
```

```
        String expectedOutputs[][] = {
                { "AACD", "CD" },
                { "ACD", "CD" } };
        return Arrays.asList(expectedOutputs);
    }

    @Test
    public void testTruncateAInFirst2Positions() {
        assertEquals(expectedOutput,
                helper.truncateAInFirst2Positions(input));
    }
}
```

Trong ví dụ trên, chúng ta sử dụng Parameterized Tests để kiểm tra phép nhân của lớp **StringHelper**. Phương thức **testConditions()** trả về một Collection chứa các mảng dữ liệu (Object[]) đại diện cho các bộ tham số. Mỗi bộ dữ liệu được truyền vào constructor của **StringHelperParameterizedTest** class. Test case **testTruncateAInFirst2Positions()** sẽ chạy với các giá trị tham số tương ứng.

Khi chạy Parameterized Tests, JUnit sẽ tạo các phiên bản của test class cho mỗi bộ dữ liệu và chạy test case tương ứng. Các kết quả được tổng hợp và báo cáo riêng rẽ cho mỗi bộ dữ liệu.

10. Organize JUnits into Suites

Để tổ chức JUnit tests vào trong bộ (suite), bạn có thể làm theo các bước sau:

1. Tạo một lớp mới và đặt tên cho lớp đó, ví dụ TestSuite.
2. Sử dụng annotation @RunWith(Suite.class) để đánh dấu lớp đó sẽ chạy như một suite.
3. Sử dụng annotation @Suite.SuiteClasses và liệt kê các lớp test case mà bạn muốn chạy trong bộ (suite).

Dưới đây là một ví dụ cụ thể:

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
        StringHelperTest.class,
        ArraysCompareTest.class,
        QuickBeforeAfterTest.class
})
public class TestSuite {
    // Không cần viết thêm mã trong lớp TestSuite
}
```

Trong ví dụ trên, chúng ta tạo ra một lớp TestSuite và sử dụng @RunWith(Suite.class) để đánh dấu rằng lớp này là một suite. Trong phần @Suite.SuiteClasses, chúng ta liệt kê các lớp test case mà chúng ta muốn chạy trong bộ (suite). Ở đây, chúng ta liệt kê StringHelperTest.class, ArraysCompareTest.class, và QuickBeforeAfterTest.class.

Sau khi bạn đã tạo và cấu hình lớp TestSuite, bạn có thể chạy suite bằng cách nhấp chuột phải vào lớp và chọn "Run 'TestSuite'" hoặc "Debug 'TestSuite'" từ menu ngữ cảnh. Điều này sẽ chạy tất cả các test case được định nghĩa trong suite một cách tự động.

Section 2: Mockito Basics

1. Mokito là gì?

Mockito là một framework hỗ trợ tạo unit test bằng cách sử dụng các đối tượng giả (Mock hay TestDouble) theo cách dễ sử dụng mà không tạo ra "nhiễu" từ các tương tác không liên quan.

Một số thuận lợi khi sử dụng Mockito:

- Đơn giản: dễ dàng tạo các đối tượng giả và giả lập kết quả, hành vi test.
- Số lượng API của Mockito không nhiều, nhưng đáp ứng đầy đủ các yêu cầu để giả lập các hành vi test.
- Tập trung vào test các hành vi cụ thể, giảm thiểu các phiền nhiễu từ các tương tác không liên quan.

2. Phân loại Mock/ Test Double

Có nhiều trường hợp sử dụng các đối tượng giả, chúng ta có thể phân loại chúng như sau:

- **Dummy object** là các đối tượng được di chuyển khắp nơi nhưng không bao được sử dụng thật sự. Thường chúng chỉ được sử dụng để truyền danh sách tham số.
- **Fake object** là các đối tượng thật đã được cài đặt một cách đầy đủ, nhưng thường được sử dụng trong môi trường test, không phù hợp cho môi trường thật.
- **Stub object** là một chương trình hoặc thành phần giả lập dùng để kiểm thử, nó cung cấp câu trả lời cho các cuộc gọi trong khi kiểm thử. Thông thường nó không đáp lại bất kỳ thứ gì ngoài những gì đã được lập trình cho kiểm thử.
- **Mock object (MO)** là một đối tượng, dùng để giả lập hành vi của các class bên ngoài để thực hiện hành vi mà chúng ta mong muốn. Những giả lập này do chúng ta quản lý nên nó đảm bảo chất lượng của những đoạn code mà chúng ta đang viết Unit Test.
- **Spy object** : là một trường hợp đặc biệt của Stub và Mock, nó có thể gọi thực sự behavior của dependency hoặc mock một số behavior nếu cần.

3. Cài đặt Mockito

Để sử dụng Mockito chúng ta cần thêm thư viện này vào project. Với project maven, chúng ta mở file pom.xml và thêm khai báo sau:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.10.19</version>
    <scope>test</scope>
</dependency>
```

4. Introduction to Argument Matchers

Trong Mockito, Argument Matchers (bộ so trùng tham số) được sử dụng để so trùng các giá trị tham số khi thiết lập hành vi giả định (stub) hoặc kiểm tra hành vi của các phương thức trong các đối tượng mock. Argument Matchers cho phép bạn chỉ định các quy tắc linh hoạt để so trùng các giá trị tham số, thay vì chỉ dựa trên các giá trị cụ thể. Dưới đây là một số khái niệm cơ bản về Argument Matchers trong Mockito:

- **any():** Đây là một Argument Matcher chấp nhận bất kỳ giá trị tham số nào cho tham số tương ứng. Nó được sử dụng để chỉ định rằng bất kỳ giá trị nào đều được chấp nhận.

```
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.when;

List<String> mockList = mock(List.class);
when(mockList.get(anyInt())).thenReturn("Mockito");

System.out.println(mockList.get(0)); // Output: "Mockito"
System.out.println(mockList.get(1)); // Output: "Mockito"
```

- **eq():** Đây là một Argument Matcher chấp nhận giá trị tham số khớp chính xác với giá trị cụ thể. Nó được sử dụng để chỉ định rằng giá trị tham số cần phải khớp chính xác với giá trị được chỉ định.

```
import static org.mockito.ArgumentMatchers.eq;
import static org.mockito.Mockito.when;

List<String> mockList = mock(List.class);
when(mockList.contains(eq("Mockito"))).thenReturn(true);

System.out.println(mockList.contains("Mockito")); // Output: true
System.out.println(mockList.contains("JUnit"));   // Output: false
```

- **anyXxx():** Mockito cung cấp các Argument Matcher được định nghĩa sẵn cho một số kiểu dữ liệu như anyInt(), anyString(), anyList(), và nhiều hơn nữa. Chúng được sử dụng để chỉ định rằng bất kỳ giá trị tham số cụ thể của kiểu tương ứng đều được chấp nhận.

```
import static org.mockito.ArgumentMatchers.anyString;
import static org.mockito.Mockito.when;

Map<String, Integer> mockMap = mock(Map.class);
when(mockMap.containsKey(anyString())).thenReturn(true);

System.out.println(mockMap.containsKey("key"));     // Output: true
System.out.println(mockMap.containsKey("another")); // Output: true
```

5. Mockito để mock (giả định) các phương thức trong đối tượng List.

```
@Test
public void letsMockListSize() {
    List list = mock(List.class);
    Mockito.when(list.size()).thenReturn(10);
    assertEquals(10, list.size());
}

@Test
public void letsMockListSizeWithMultipleReturnValues() {
    List list = mock(List.class);
    Mockito.when(list.size()).thenReturn(10).thenReturn(20);
    assertEquals(10, list.size()); // First Call
    assertEquals(20, list.size()); // Second Call
}

@Test
public void letsMockListGet() {
    List<String> list = mock(List.class);
    Mockito.when(list.get(0)).thenReturn("in28Minutes");
    assertEquals("in28Minutes", list.get(0));
    assertNull(list.get(1));
}

@Test
public void letsMockListGetWithAny() {
    List<String> list = mock(List.class);
    Mockito.when(list.get(Mockito.anyInt())).thenReturn("in28Minutes");
    // If you are using argument matchers, all arguments
    // have to be provided by matchers.
    assertEquals("in28Minutes", list.get(0));
    assertEquals("in28Minutes", list.get(1));
}
```

letsMockListSize():

- Chúng ta tạo một đối tượng mock của List bằng cách sử dụng mock(List.class).
- Sử dụng Mockito.when().thenReturn() để giả định rằng khi gọi phương thức size() trên đối tượng mock, nó sẽ trả về giá trị là 10.
- Sử dụng assertEquals() để kiểm tra xem phương thức size() trên đối tượng mock trả về giá trị là 10.

letsMockListSizeWithMultipleReturnValues():

- Tương tự như letsMockListSize(), nhưng chúng ta sử dụng thenReturn() nhiều lần để giả định nhiều giá trị trả về.
- Khi gọi phương thức size() lần đầu tiên, nó sẽ trả về 10. Khi gọi lần thứ hai, nó sẽ trả về 20.
- Sử dụng assertEquals() để kiểm tra kết quả trả về của size() theo từng lần gọi.

letsMockListGet():

- Chúng ta tạo một đối tượng mock của List với kiểu dữ liệu là String (List<String>).
- Sử dụng Mockito.when().thenReturn() để giả định rằng khi gọi phương thức get(0) trên đối tượng mock, nó sẽ trả về chuỗi "in28Minutes".

- Sử dụng assertEquals() để kiểm tra xem phương thức get(0) trên đối tượng mock trả về chuỗi "in28Minutes".
- Sử dụng assertNull() để kiểm tra xem phương thức get(1) trên đối tượng mock trả về giá trị null.

letsMockListGetWithAny():

- Tương tự như letsMockListGet(), nhưng chúng ta sử dụng Mockito.anyInt() để giả định rằng khi gọi phương thức get() với bất kỳ tham số nào kiểu int, nó sẽ trả về chuỗi "in28Minutes".
- Sử dụng Mockito.anyInt() để so trùng bất kỳ giá trị tham số nguyên nào trong giả định.
- Sử dụng assertEquals() để kiểm tra xem phương thức get(0) và get(1) trên đối tượng mock trả về cùng chuỗi "in28Minutes".

Trong tất cả các ví dụ trên, chúng ta sử dụng Mockito để tạo các đối tượng mock của List và giả định hành vi của các phương thức trong đối tượng mock. Sau đó, chúng ta sử dụng các phương thức assertEquals() hoặc assertNull() để kiểm tra kết quả trả về từ các phương thức được giả định.

6. Examples mocking List class

TodoService

```
import java.util.List;

// External Service - Lets say this comes from WunderList
public interface TodoService {
    public List<String> retrieveTodos(String user);
}
```

TodoBusinessImpl

```
public class TodoBusinessImpl {
    private TodoService todoService;

    TodoBusinessImpl(TodoService todoService) {
       this.todoService = todoService;
    }

    public List<String> retrieveTodosRelatedToSpring(String user) {
       List<String> filteredTodos = new ArrayList<String>();
       List<String> allTodos = todoService.retrieveTodos(user);
       for (String todo : allTodos) {
          if (todo.contains("Spring")) {
             filteredTodos.add(todo);
          }
       }
       return filteredTodos;
    }
}
```

TodoBusinessImplMockitoTest

```
public class TodoBusinessImplMockitoTest {

    @Test
    public void usingMockito() {
```

```
        TodoService todoService = mock(TodoService.class);
        List<String> allTodos = Arrays.asList("Learn Spring MVC",
                "Learn Spring", "Learn to Dance");
        when(todoService.retrieveTodos("Ranga")).thenReturn(allTodos);
        TodoBusinessImpl todoBusinessImpl = new TodoBusinessImpl(todoService);
        List<String> todos = todoBusinessImpl
                .retrieveTodosRelatedToSpring("Ranga");
        assertEquals(2, todos.size());
    }
}
```

Trong lớp TodoBusinessImplMockitoTest, chúng ta sử dụng Mockito để kiểm thử đơn vị (unit test) cho TodoBusinessImpl. Trong phương thức usingMockito(), chúng ta tạo một mock object của TodoService bằng cách sử dụng mock(TodoService.class). Chúng ta cũng định nghĩa hành vi giả định cho mock object bằng cách sử dụng when().thenReturn().

Trong trường hợp này, chúng ta giả định rằng khi gọi retrieveTodos("Ranga") trên mock object todoService, nó sẽ trả về danh sách công việc được cung cấp. Sau đó, chúng ta tạo một đối tượng TodoBusinessImpl với mock object todoService và gọi phương thức retrieveTodosRelatedToSpring("Ranga").

Cuối cùng, chúng ta sử dụng phương thức assertEquals() để so sánh kết quả trả về từ retrieveTodosRelatedToSpring() với kết quả mong đợi (danh sách có 2 công việc liên quan đến "Spring"). Nếu kết quả trả về khớp với kết quả mong đợi, test case sẽ thành công.

Mockito giúp chúng ta tạo các đối tượng mock để thay thế các thành phần phụ thuộc và kiểm tra hành vi của lớp cần được kiểm thử mà không cần phụ thuộc vào các thành phần thực tế. Trong trường hợp này, chúng ta mock TodoService để kiểm tra xem phương thức retrieveTodosRelatedToSpring() trong TodoBusinessImpl hoạt động như mong đợi khi được cung cấp các công việc từ TodoService.

7. BDD Style- Given When Then

Giới thiệu về BDD

BDD (Behavior-Driven Development) là một phương pháp phát triển phần mềm tập trung vào việc mô tả hành vi của hệ thống thông qua các kịch bản được viết bằng ngôn ngữ tự nhiên gần gũi với ngôn ngữ của người không chuyên. Mockito là một framework được sử dụng phổ biến trong việc kiểm thử đơn vị (unit testing) trong Java.

Trong Mockito, BDD Style là một phong cách viết kiểm thử dựa trên BDD, giúp cung cấp các phương pháp và cú pháp dễ đọc, dễ hiểu hơn cho việc viết kiểm thử. Nó giúp người viết kiểm thử tập trung vào hành vi của hệ thống và mô tả các kịch bản kiểm thử theo cách gần gũi với ngôn ngữ tự nhiên.

Trong Mockito, để sử dụng BDD Style, chúng ta cần sử dụng thư viện BDDMockito. Thư viện này cung cấp các phương thức và cú pháp cho việc viết kiểm thử theo phong cách BDD. Dưới đây là một ví dụ minh họa về việc sử dụng BDD Style trong Mockito:

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

// Tạo mock object
        List<String> mockList = mock(List.class);

// Định nghĩa hành vi của mock object
        given(mockList.get(0)).willReturn("Mockito");

// Thực hiện kiểm thử
        String result = mockList.get(0);

// Kiểm tra kết quả
        assertThat(result).isEqualTo("Mockito");

// Kiểm tra hành vi đã được gọi
        verify(mockList).get(0);
```

Trong ví dụ trên, chúng ta sử dụng phương thức given() để định nghĩa hành vi của mock object. Trong trường hợp này, khi gọi mockList.get(0), mock object sẽ trả về giá trị "Mockito". Chúng ta cũng sử dụng phương thức willReturn() để chỉ định giá trị trả về.

Sau đó, chúng ta thực hiện kiểm tra bằng cách gọi mockList.get(0) và so sánh kết quả với giá trị mong đợi bằng phương thức assertThat(). Cuối cùng, chúng ta kiểm tra xem phương thức get(0) đã được gọi bằng cách sử dụng phương thức verify().

Việc sử dụng BDD Style trong Mockito giúp viết kiểm thử dễ đọc, dễ hiểu và cung cấp một cú pháp gần gũi với ngôn ngữ tự nhiên, giúp tăng tính tương tác và sự thống nhất trong đội ngũ phát triển phần mềm.

Given When Then

Given-When-Then là một kỹ thuật phát triển phần mềm trong phương pháp kiểm thử hành vi (Behavior-Driven Development - BDD). Nó cung cấp một cách tổ chức và mô tả các kịch bản kiểm thử một cách rõ ràng và dễ hiểu.

Cấu trúc của Given-When-Then bao gồm:

- Given (Điều kiện ban đầu): Định nghĩa các điều kiện ban đầu, trạng thái hoặc môi trường để thực hiện kiểm thử.
- When (Hành động): Thực hiện một hành động cụ thể hoặc gọi một phương thức trong hệ thống.
- Then (Kết quả): Kiểm tra kết quả hoặc hành vi mong đợi từ hành động đã thực hiện. Đây là nơi mô tả các điều kiện hoặc giả định về kết quả và kiểm tra xem kết quả thực tế có khớp với kết quả mong đợi hay không.

Trong Mockito, chúng ta có thể sử dụng phong cách Given-When-Then để viết các kiểm thử sử dụng framework Mockito. Dưới đây là một ví dụ minh họa:

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

public class ExampleTest {
    @Test
    public void testExample() {
        // Given
        SomeClass mockObject = mock(SomeClass.class);
        when(mockObject.someMethod()).thenReturn("Mockito");

        // When
        String result = mockObject.someMethod();

        // Then
        assertEquals(result,"Mockito");
        verify(mockObject).someMethod();
    }
    @Test
    public void testExample1() {
        // Given
        SomeClass mockObject = mock(SomeClass.class);
        given(mockObject.someMethod()).willReturn("Mockito");

        // When
        String result = mockObject.someMethod();

        // Then
        assertEquals(result,"Mockito");
        verify(mockObject).someMethod();
    }

}
```

Trong ví dụ trên, chúng ta sử dụng Mockito để tạo một mock object của lớp SomeClass. Trong phần Given, chúng ta sử dụng phương thức when() và thenReturn() để định nghĩa hành vi của mock object. Chúng ta chỉ định rằng khi gọi mockObject.someMethod(), nó sẽ trả về giá trị "Mockito".

Trong phần When, chúng ta gọi mockObject.someMethod() và lưu kết quả vào biến result.

Cuối cùng, trong phần Then, chúng ta sử dụng các phương thức kiểm tra của framework kiểm thử (ví dụ: assertEquals() để kiểm tra kết quả có khớp với giá trị mong đợi hay không. Chúng ta cũng sử dụng phương thức verify() để kiểm tra xem phương thức someMethod() đã được gọi hay chưa.

Phong cách Given-When-Then giúp tạo ra các kịch bản kiểm thử dễ đọc, dễ hiểu và dễ duy trì, đồng thời tập trung vào hành vi của hệ thống và các kết quả mong đợi.

BDD Mockito Syntax

Trong Mockito, để sử dụng cú pháp BDD Style, chúng ta cần sử dụng thư viện BDDMockito. Dưới đây là một số cú pháp quan trọng khi sử dụng BDD Style trong Mockito:

- **given()** - Định nghĩa hành vi của mock object:
```
given(mockObject.methodCall()).willReturn(result);
```
Ví dụ: given(mockList.get(0)).willReturn("Mockito");

- willReturn() - Chỉ định giá trị trả về khi gọi phương thức của mock object:
```
given(mockObject.methodCall()).willReturn(result);
```
Ví dụ: given(mockList.get(0)).willReturn("Mockito");
- willThrow() - Chỉ định ngoại lệ sẽ được ném ra khi gọi phương thức của mock object:
```
given(mockObject.methodCall()).willThrow(exception);
```
Ví dụ: given(mockList.get(0)).willThrow(new RuntimeException());
- willAnswer() - Chỉ định một hành động tùy chỉnh khi gọi phương thức của mock object:
```
given(mockObject.methodCall()).willAnswer(invocation ->
customAction(invocation.getArgument(0)));
```
Ví dụ
```
given(mockList.get(anyInt())).willAnswer(invocation -> {
    int index = invocation.getArgument(0);
    return "Element " + index;
    });
```
- verify() - Kiểm tra xem phương thức đã được gọi hay chưa:
```
verify(mockObject).methodCall();
Ví dụ: verify(mockList).get(0);
```
- verify() với số lần gọi:
```
verify(mockObject, times(n)).methodCall();
Ví dụ: verify(mockList, times(2)).add("Element");
```
- verify() với tham số cụ thể:
```
verify(mockObject).methodCall(argument);
Ví dụ: verify(mockList).add("Element");
```
- verifyNoMoreInteractions() - Kiểm tra xem đã không còn tương tác nào khác với mock object hay không:
```
verifyNoMoreInteractions(mockObject);
```

8. Ví dụ về BDD Mockito

```
public class TodoBusinessImplMockitoTest {

    @Test
    public void usingMockito_UsingBDD() {
       TodoService todoService = mock(TodoService.class);
       TodoBusinessImpl todoBusinessImpl = new TodoBusinessImpl(todoService);
       List<String> allTodos = Arrays.asList("Learn Spring MVC",
             "Learn Spring", "Learn to Dance");

       //given
       given(todoService.retrieveTodos("Ranga")).willReturn(allTodos);

       //when
       List<String> todos = todoBusinessImpl
             .retrieveTodosRelatedToSpring("Ranga");

       //then
       assertThat(todos.size(), is(2));
    }

}
```

Đoạn code trên là một unit test sử dụng thư viện Mockito và thực hiện kiểm tra logic xử lý của phương thức retrieveTodosRelatedToSpring trong lớp TodoBusinessImpl.

Cụ thể, đoạn code này sử dụng phong cách BDD (Behavior Driven Development) để viết test case. BDD là một phương pháp phát triển phần mềm tập trung vào hành vi của hệ thống, sử dụng ngôn ngữ tự nhiên để miêu tả và xác định hành vi đó.

Dưới đây là giải thích từng phần của đoạn code:

- TodoService todoService = mock(TodoService.class);: Đoạn code này tạo một mock object của lớp TodoService bằng cách sử dụng Mockito.
- TodoBusinessImpl todoBusinessImpl = new TodoBusinessImpl(todoService);: Tạo một đối tượng TodoBusinessImpl với mock object todoService đã được tạo ở bước trước.
- List<String> allTodos = Arrays.asList("Learn Spring MVC", "Learn Spring", "Learn to Dance");: Tạo một danh sách các công việc (allTodos) mô phỏng là danh sách công việc trả về từ todoService.retrieveTodos("Ranga").
- given(todoService.retrieveTodos("Ranga")).willReturn(allTodos);: Đoạn code này sử dụng phương thức given() của BDDMockito để định nghĩa hành vi của mock object todoService. Nó chỉ định rằng khi gọi phương thức retrieveTodos("Ranga") trên todoService, nó sẽ trả về danh sách allTodos đã được định nghĩa ở trên.
- List<String> todos = todoBusinessImpl.retrieveTodosRelatedToSpring("Ranga");: Gọi phương thức retrieveTodosRelatedToSpring() trên đối tượng todoBusinessImpl để kiểm tra.
- assertThat(todos.size(), is(2));: Đoạn code này sử dụng phương thức assertThat() của framework kiểm thử để kiểm tra xem kích thước của danh sách todos có phải là 2 hay không.

Tổng quan, đoạn code trên sử dụng Mockito để tạo mock object của TodoService, định nghĩa hành vi của mock object đối với phương thức retrieveTodos(), và sau đó kiểm tra phương thức retrieveTodosRelatedToSpring() trong TodoBusinessImpl để đảm bảo rằng nó hoạt động như mong đợi.

9. Verify calls on Mocks

Phương thức verify() trong Mockito được sử dụng để kiểm tra xem một phương thức đã được gọi hay chưa trên một mock object. Việc sử dụng verify() có một số lợi ích quan trọng:

- Xác minh hành vi: Sử dụng verify(), bạn có thể xác minh rằng phương thức kiểm thử thực sự gọi các phương thức chính xác trên mock object theo dự định. Điều này đảm bảo rằng các phương thức đang được kiểm thử hoạt động như mong đợi và tuân thủ các luồng điều khiển và quy tắc kinh doanh.
- Đảm bảo tính đúng đắn: Việc xác minh giúp đảm bảo tính đúng đắn của hệ thống bằng cách kiểm tra rằng các phương thức và hành vi đã được triển khai và sử dụng đúng cách. Điều này giúp tránh việc phát hiện lỗi sau này hoặc các hành vi không mong muốn trong quá trình phát triển và triển khai.
- Kiểm tra các luồng điều khiển: Khi sử dụng verify() cùng với các phương thức xác minh như times() hoặc inOrder(), bạn có thể kiểm tra xem các phương thức đã được gọi theo đúng thứ tự và số lượng như mong đợi. Điều này giúp đảm bảo rằng các luồng điều khiển của hệ thống hoạt động đúng và không có sai sót nào.
- Hỗ trợ trong gỡ lỗi: Trong quá trình gỡ lỗi, việc sử dụng verify() có thể giúp xác định xem một phương thức nào đã được gọi và với các đối số nào. Điều này giúp cung cấp thông tin quan trọng để xác định nguyên nhân của các lỗi hoặc hành vi không mong muốn trong hệ thống.

Vì vậy, việc sử dụng verify() trong Mockito là một công cụ quan trọng để kiểm tra và đảm bảo tính chính xác và đúng đắn của hệ thống khi thực hiện kiểm thử.

Dưới đây là cách bạn có thể xác minh cuộc gọi trên một mock:

❖ **Xác minh xem một phương thức đã được gọi:**

```
verify(mockObject).methodName();
```

Điều này xác minh rằng phương thức methodName() đã được gọi trên mockObject.

❖ **Xác minh xem một phương thức đã được gọi một số lần cụ thể:**

```
verify(mockObject, times(n)).methodName();
```

Điều này xác minh rằng phương thức methodName() đã được gọi chính xác n lần trên mockObject.

❖ **Xác minh xem một phương thức không bao giờ được gọi:**

```
verify(mockObject, never()).methodName();
```

Điều này xác minh rằng phương thức methodName() không bao giờ được gọi trên mockObject.

❖ **Xác minh xem một phương thức đã được gọi với các đối số cụ thể:**

```
verify(mockObject).methodName(arg1, arg2);
```

Điều này xác minh rằng phương thức methodName() đã được gọi trên mockObject với các đối số cụ thể arg1 và arg2.

❖ **Xác minh xem một phương thức đã được gọi theo một thứ tự cụ thể:**

```
InOrder inOrder = inOrder(mockObject);

inOrder.verify(mockObject).method1();

inOrder.verify(mockObject).method2();
```

Điều này xác minh rằng method1() đã được gọi trước method2() trên mockObject.

Đây là một số tùy chọn xác minh thông thường được cung cấp bởi Mockito. Bạn có thể tùy chỉnh quá trình xác minh dựa trên yêu cầu cụ thể của bạn. Mockito cũng cung cấp các phương thức xác minh bổ sung để xử lý các tình huống phức tạp hơn. Tham khảo tài liệu Mockito để biết thêm thông tin về các tùy chọn xác minh nâng cao.

Ví dụ:

```
@Test
public void letsTestDeleteNow() {

    TodoService todoService = mock(TodoService.class);

    List<String> allTodos = Arrays.asList("Learn Spring MVC",
          "Learn Spring", "Learn to Dance");

    when(todoService.retrieveTodos("Ranga")).thenReturn(allTodos);

    TodoBusinessImpl todoBusinessImpl = new TodoBusinessImpl(todoService);

    todoBusinessImpl.deleteTodosNotRelatedToSpring("Ranga");

    verify(todoService).deleteTodo("Learn to Dance");

    verify(todoService, Mockito.never()).deleteTodo("Learn Spring MVC");

    verify(todoService, Mockito.never()).deleteTodo("Learn Spring");

    verify(todoService, Mockito.times(1)).deleteTodo("Learn to Dance");
    // atLeastOnce, atLeast

}
```

10. Capturing arguments passed to a Mock (Step08.md)

Để bắt giữ một đối số được truyền vào một mock object trong Mockito, bạn có thể sử dụng lớp ArgumentCaptor. ArgumentCaptor cho phép bạn bắt giữ và lấy giá trị của đối số được truyền vào một mock trong quá trình gọi phương thức. Dưới đây là cách bạn có thể bắt giữ một đối số bằng Mockito:

- **Tạo một instance của lớp ArgumentCaptor, chỉ định kiểu của đối số bạn muốn bắt giữ:**

```
ArgumentCaptor<LoaiDoiSo> argumentCaptor = ArgumentCaptor.forClass(LoaiDoiSo.class);
```

- **Sử dụng phương thức verify() để bắt giữ đối số khi phương thức được gọi trên mock object, và truyền ArgumentCaptor làm đối số:**

```
verify(mockObject).methodName(argumentCaptor.capture());
```

- Lấy giá trị của đối số đã bắt giữ bằng phương thức getValue() của ArgumentCaptor:

```
LoaiDoiSo doiSoDaBat = argumentCaptor.getValue();
```

Dưới đây là một ví dụ để minh họa cách bắt giữ đối số được truyền vào một mock object:

```
public class ExampleTest {
    @Test
    public void testExample() {
        SomeClass mockObject = mock(SomeClass.class);
        ArgumentCaptor<String> argumentCaptor =
ArgumentCaptor.forClass(String.class);

        mockObject.someMethod("Mockito");

        verify(mockObject).someMethod(argumentCaptor.capture());

        String doiSoDaBat = argumentCaptor.getValue();
        assertEquals(doiSoDaBat, "Mockito");
    }
}
```

Trong ví dụ này, chúng ta tạo một mock object của SomeClass và một ArgumentCaptor cho kiểu String. Sau đó, khi gọi someMethod("Mockito") trên mock object, chúng ta bắt giữ đối số bằng cách sử dụng verify() và ArgumentCaptor. Cuối cùng, chúng ta lấy giá trị của đối số đã bắt giữ bằng getValue() và thực hiện các phép xác minh.

Bắt giữ đối số sử dụng ArgumentCaptor cho phép bạn xem xét và xác minh các giá trị được truyền vào một mock object trong quá trình gọi phương thức, mang lại tính linh hoạt và kiểm soát hơn trong kiểm thử của bạn.

Section 3: Mockito Advanced

1. Hamcrest Matchers (Step9)

Hamcrest Matchers là một thư viện cung cấp một tập hợp các phương thức kiểm tra linh hoạt và dễ đọc trong việc so sánh giá trị trong kiểm thử đơn vị (unit testing) trong Java. Thư viện này giúp tạo ra các câu khẳng định (assertions) mạnh mẽ và dễ đọc hơn, giúp tăng tính rõ ràng và tường minh của các kiểm tra.

Dưới đây là một số điểm chính của Hamcrest Matchers:

- **Cú pháp đọc dễ hiểu:** Hamcrest Matchers sử dụng cú pháp đọc gần gũi với ngôn ngữ tự nhiên, giúp làm cho các câu khẳng định trong kiểm thử trở nên dễ hiểu và rõ ràng hơn.
- **Kiểm tra linh hoạt:** Thư viện cung cấp một loạt các phương thức kiểm tra, từ các phương thức cơ bản như equalTo() và is() cho đến các phương thức phức tạp hơn như hasItem(), containsString(), greaterThan(), vv. Điều này cho phép bạn kiểm tra các điều kiện phù hợp với nhu cầu kiểm thử cụ thể của bạn.
- **Kết hợp linh hoạt:** Hamcrest Matchers cho phép bạn kết hợp các kiểm tra lại với nhau để tạo ra các câu khẳng định phức tạp hơn. Bạn có thể sử dụng các phương thức như allOf(), anyOf(), not(), vv. để kết hợp các kiểm tra thành các biểu thức điều kiện phức tạp.
- **Tích hợp với các framework kiểm thử:** Hamcrest Matchers tích hợp tốt với các framework kiểm thử phổ biến như JUnit và Mockito, giúp tạo ra các câu khẳng định mạnh mẽ và dễ đọc trong quá trình viết kiểm thử.

Dưới đây là một ví dụ minh họa về việc sử dụng Hamcrest Matchers:

```
import static org.hamcrest.MatcherAssert.*;
import static org.hamcrest.Matchers.*;

public class ExampleTest {
    @Test
    public void testExample() {
        String value = "Hello World";

        assertThat(value, is(equalTo("Hello World")));
        assertThat(value, containsString("Hello"));
        assertThat(value.length(), greaterThan(5));
    }
}
```

Trong ví dụ trên, chúng ta sử dụng Hamcrest Matchers để so sánh và kiểm tra giá trị value. Chúng ta sử dụng các phương thức như is(), equalTo(), containsString(), greaterThan() để tạo ra các câu khẳng định mạnh mẽ và dễ đọc.

Hamcrest Matchers giúp tạo ra các câu khẳng định linh hoạt và dễ đọc hơn trong việc kiểm thử đơn vị, giúp tăng tính rõ ràng và tường minh trong việc xác nhận các điều kiện và giá trị trong quá trình kiểm thử.

Đoạn mã trên là một ví dụ về việc sử dụng các Hamcrest Matchers trong kiểm thử.

Kiểm tra List:

```
List<Integer> scores = Arrays.asList(99, 100, 101, 105);
assertThat(scores, hasSize(4));
assertThat(scores, hasItems(100, 101));
assertThat(scores, everyItem(greaterThan(90)));
assertThat(scores, everyItem(lessThan(200)));
```

- Phần này kiểm tra một danh sách số nguyên scores.
- hasSize(4) xác minh rằng danh sách có kích thước là 4.
- hasItems(100, 101) xác minh rằng danh sách chứa các phần tử 100 và 101.
- everyItem(greaterThan(90)) xác minh rằng tất cả các phần tử trong danh sách đều lớn hơn 90.
- **everyItem(lessThan(200)) xác minh rằng tất cả các phần tử trong danh sách đều nhỏ hơn 200.**

Kiểm tra String:

```
// String
assertThat("", isEmptyString());
assertThat(null, isEmptyOrNullString());
```

- Phần này kiểm tra chuỗi.
- isEmptyString() xác minh rằng chuỗi là rỗng.
- isEmptyOrNullString() xác minh rằng chuỗi là rỗng hoặc null.

Kiểm tra mảng:

```
// Array
Integer[] marks = { 1, 2, 3 };

assertThat(marks, arrayWithSize(3));
assertThat(marks, arrayContainingInAnyOrder(2, 3, 1));
```

- Phần này kiểm tra một mảng số nguyên marks.
- arrayWithSize(3) xác minh rằng mảng có kích thước là 3.
- arrayContainingInAnyOrder(2, 3, 1) xác minh rằng mảng chứa các phần tử 2, 3 và 1, không quan tâm đến thứ tự.

Các câu lệnh assertThat() sử dụng Hamcrest Matchers để tạo ra các câu khẳng định mạnh mẽ và dễ đọc trong quá trình kiểm thử. Các Hamcrest Matchers được sử dụng để kiểm tra các điều kiện và giá trị khác nhau, từ kích thước của danh sách, phần tử có trong danh sách, đến tính chất của chuỗi và mảng.

2. Mockito Annotations. @Mock, @InjectMocks, @RunWith(MockitoJUnitRunner.class), @Captor (Step 10)

Trong Mockito, các chú thích (annotations) được sử dụng để giả định và quản lý các mock object và các phụ thuộc trong quá trình kiểm thử. Dưới đây là giải thích cho một số chú thích Mockito phổ biến:

- **@Mock:** Chú thích @Mock được sử dụng để tạo một mock object cho một lớp hoặc giao diện. Khi bạn chú thích một trường (field) bằng @Mock, Mockito sẽ tạo một đối tượng mock của lớp hoặc giao diện tương ứng.
- **@InjectMocks:** Chú thích @InjectMocks được sử dụng để tự động tiêm các mock object vào đối tượng mà bạn muốn kiểm thử. Khi bạn chú thích một đối tượng bằng @InjectMocks, Mockito sẽ cố gắng tiêm các mock object đã được tạo vào các trường (fields) tương ứng của đối tượng đó.
- **@RunWith(MockitoJUnitRunner.class):** Chú thích @RunWith(MockitoJUnitRunner.class) được sử dụng để chỉ định lớp MockitoJUnitRunner như một Runner cho các kiểm thử JUnit. Khi bạn chú thích một lớp kiểm thử bằng @RunWith(MockitoJUnitRunner.class), MockitoJUnitRunner sẽ thực thi các kiểm thử và tự động quản lý các mock object và phụ thuộc tương ứng.
- **@Captor:** Chú thích @Captor được sử dụng để bắt giữ (capture) các đối số được truyền vào trong một phương thức gọi của mock object. Khi bạn chú thích một trường (field) bằng @Captor và sử dụng ArgumentCaptor để bắt giữ các đối số, Mockito sẽ tự động gán đối tượng ArgumentCaptor cho trường đó.

Dưới đây là một ví dụ minh họa về việc sử dụng các chú thích Mockito:

```
@RunWith(MockitoJUnitRunner.class)
public class ExampleTest {
    @Mock
    private SomeDependency mockDependency;

    @InjectMocks
    private SomeClass classUnderTest;

    @Captor
    private ArgumentCaptor<String> stringCaptor;

    @Test
    public void testExample() {
       MockitoAnnotations.initMocks(this);

       // Use mockDependency and classUnderTest in the test

       // Verify method calls
       verify(mockDependency).someMethod(stringCaptor.capture());
       String capturedArgument = stringCaptor.getValue();
       // Perform assertions on capturedArgument
    }
}
```

Trong ví dụ trên, chúng ta sử dụng các chú thích Mockito như sau:

- @Mock được sử dụng để tạo một mock object (mockDependency) cho SomeDependency.
- @InjectMocks được sử dụng để tự động tiêm mock object vào đối tượng classUnderTest.
- @Captor được sử dụng để bắt giữ các đối số được truyền vào trong một phương thức gọi.

- @RunWith(MockitoJUnitRunner.class) được sử dụng để sử dụng MockitoJUnitRunner như một Runner cho các kiểm thử JUnit.

Các chú thích Mockito giúp giả định và quản lý các mock object và các phụ thuộc trong quá trình kiểm thử, làm cho việc viết và quản lý các kiểm thử dễ dàng và tiện lợi hơn.

3. Mockito JUnit Rule

Trong JUnit, bạn chỉ có thể sử dụng một chú thích @RunWith duy nhất để chỉ định test runner cho lớp kiểm thử của bạn. Nếu bạn sử dụng @RunWith(MockitoJUnitRunner.class), điều này có nghĩa là bạn đang sử dụng MockitoJUnitRunner làm test runner, cung cấp tích hợp giữa JUnit và Mockito.

Tuy nhiên, việc sử dụng @RunWith(MockitoJUnitRunner.class) giới hạn khả năng sử dụng các tính năng hoặc runners khác của JUnit mà có thể cần thiết cho các kiểm thử của bạn. Ví dụ, nếu bạn cần sử dụng một runner khác hoặc mở rộng một lớp cơ sở đã có runner được chỉ định, bạn không thể sử dụng @RunWith(MockitoJUnitRunner.class).

Để vượt qua giới hạn này, Mockito cung cấp phương thức MockitoJUnit.rule(), cho phép bạn sử dụng Mockito JUnit Rule. Mockito JUnit Rule có thể được thêm làm một trường trong lớp kiểm thử của bạn bằng cách sử dụng chú thích @Rule, như đã được thể hiện trong các ví dụ trước đó. Điều này cho phép bạn tích hợp khả năng mocking của Mockito với JUnit, mà không bị giới hạn vào một test runner duy nhất.

Sử dụng MockitoJUnit.rule() cung cấp tính linh hoạt hơn và cho phép bạn kết hợp các tính năng mocking của Mockito với các tính năng hoặc runners khác của JUnit theo nhu cầu. Điều này cho phép bạn lựa chọn test runner phù hợp nhất cho lớp kiểm thử của bạn trong khi vẫn sử dụng được sức mạnh của khả năng mocking của Mockito.

Trong Mockito, Rule được sử dụng để mở rộng khả năng của các framework kiểm thử như JUnit và TestNG. Rule cho phép thay đổi hoặc mở rộng cách mà các kiểm thử được thực thi, bằng cách áp dụng các quy tắc và hành động bổ sung vào quy trình kiểm thử.

JUnit Rule được sử dụng để kết hợp với Mockito và tự động quản lý các mock object và phụ thuộc. Điều này giúp việc tạo và quản lý mock object trở nên dễ dàng hơn, mà không cần sử dụng các chú thích Mockito riêng lẻ.

Dưới đây là một ví dụ minh họa về cách sử dụng Mockito JUnit Rule:

```java
import org.junit.Rule;
import org.junit.Test;
import org.mockito.Mock;
import org.mockito.junit.MockitoJUnit;
import org.mockito.junit.MockitoRule;

import static org.mockito.Mockito.*;

public class ExampleTest {
    @Mock
    private SomeDependency mockDependency;

    @Rule
    public MockitoRule mockitoRule = MockitoJUnit.rule();

    @Test
    public void testExample() {
       // Use mockDependency in the test

       // Perform verifications
       verify(mockDependency).someMethod();
    }
}
```

Trong ví dụ trên:

- @Mock được sử dụng để tạo một mock object (mockDependency) cho SomeDependency.
- @Rule được sử dụng để khai báo một Rule MockitoJUnit, đại diện cho mockitoRule.
- Trong phương thức kiểm thử testExample(), bạn có thể sử dụng mockDependency như bình thường.

Quy tắc MockitoJUnit sẽ tự động quản lý việc tạo và cấu hình các mock object. Bạn không cần thực hiện bất kỳ công việc cấu hình bổ sung nào và các mock object sẽ được sử dụng trong quá trình kiểm thử.

Sử dụng Mockito JUnit Rule giúp tạo và quản lý các mock object trở nên đơn giản hơn, giảm thiểu sự lặp lại và tăng tính tự động hoá trong quá trình kiểm thử.

4. Mockito Spy

Trong Mockito, Spy là một đối tượng bao bọc (wrapper) xung quanh một đối tượng thực tế (real object) cho phép bạn theo dõi và mô phỏng một phần hành vi của nó. Nó tương tự như một đối tượng Mock, nhưng Spy giữ lại hành vi ban đầu của đối tượng thực tế trừ khi được chỉ định khác. Khi sử dụng Spy, bạn có thể gọi các phương thức thực tế trên đối tượng và bạn có khả năng stub hoặc xác minh các phương thức cụ thể theo yêu cầu.

Khi sử dụng Spy, các phương thức thực tế của đối tượng vẫn được gọi và được thực thi, trừ khi bạn chỉ định các hành vi cụ thể cho phương thức đó.

Để tạo một Spy, bạn có thể sử dụng phương thức spy() của lớp org.mockito.Mockito. Dưới đây là cú pháp cơ bản:

```
RealObject realObject = new RealObject();
RealObject spyObject = spy(realObject);
```

Sau khi tạo ra một Spy, bạn có thể sử dụng các phương thức Mockito như when(), thenReturn(), verify() và các matcher khác để xác định hành vi cụ thể cho các phương thức của Spy. Điều này cho phép bạn kiểm tra và can thiệp vào hành vi của đối tượng thực tế.

- **Ghi đè các phương thức cụ thể trong Spy:**

Khi sử dụng Spy, bạn có thể ghi đè các phương thức cụ thể để thay đổi hành vi của chúng. Điều này cho phép bạn stub hoặc mock các phương thức đó trong Spy trong khi vẫn giữ lại hành vi ban đầu của các phương thức khác. Dưới đây là cách bạn có thể ghi đè các phương thức cụ thể trong Spy:

```
RealObject realObject = new RealObject();
RealObject spyObject = spy(realObject);

// Ghi đè hành vi của phương thức cụ thể
when(spyObject.tenPhuongThuc()).thenReturn(giaTriMongMuon);
```

Dưới đây là một ví dụ minh họa về việc sử dụng Spy trong Mockito:

```
public class ExampleTest {
    @Test
    public void testSpy() {
        List<String> list = new ArrayList<>();
        List<String> spyList = spy(list);

        spyList.add("Mockito");
        spyList.add("Spy");

        when(spyList.size()).thenReturn(10);

        assertThat(spyList.get(0)).isEqualTo("Mockito");
        assertThat(spyList.get(1)).isEqualTo("Spy");
        assertThat(spyList.size()).isEqualTo(10);

        verify(spyList).add("Mockito");
        verify(spyList).add("Spy");
    }
}
```

Trong ví dụ này, chúng ta tạo một Spy từ một đối tượng ArrayList. Chúng ta thêm các phần tử vào Spy và sử dụng when() và thenReturn() để xác định kết quả trả về cho phương thức size(). Cuối cùng, chúng ta kiểm tra các giá trị của Spy và sử dụng verify() để xác minh rằng các phương thức đã được gọi như mong đợi.

Qua đó, Mockito Spy cho phép bạn theo dõi và can thiệp vào đối tượng thực tế, giúp kiểm tra và điều chỉnh hành vi của đối tượng trong quá trình kiểm thử.

Lưu ý:

Trong Mockito, việc tạo đối tượng bằng spy() và mock() có những khác biệt quan trọng như sau:

- **spy():** Phương thức spy() được sử dụng để tạo một Spy object bằng việc bao bọc (wrap) một đối tượng thực tế (real object) đã tồn tại. Điều này có nghĩa là Spy sẽ theo dõi và mô phỏng một phần hành vi của đối tượng thực tế, trong khi giữ lại hành vi ban đầu của các phương thức thực tế. Khi sử dụng Spy, bạn có thể gọi các phương thức thực tế, ghi đè hành vi của các phương thức cụ thể và xác minh hoặc stub các phương thức theo yêu cầu.
- **mock():** Phương thức mock() được sử dụng để tạo một Mock object mới, đại diện cho một lớp hoặc giao diện. Mock object này không phụ thuộc vào đối tượng thực tế nào và không thực hiện bất kỳ hành vi thực tế nào của lớp gốc. Bạn chỉ có thể định nghĩa hành vi cho các phương thức được gọi trên Mock object và xác minh cách mà nó được sử dụng trong quá trình kiểm thử.

Tóm lại, sự khác biệt quan trọng giữa Spy và Mock object là Spy bao gồm một đối tượng thực tế và giữ lại hành vi của nó, trong khi Mock object không phụ thuộc vào đối tượng thực tế và chỉ chú trọng đến việc ghi đè hành vi và kiểm tra cách sử dụng nó. Việc sử dụng Spy và Mock object phụ thuộc vào mục tiêu kiểm thử của bạn và yêu cầu cụ thể của từng trường hợp.

5. Why does Mockito not allow stubbing final & private methods?

Mockito không cho phép stubbing (định nghĩa hành vi giả định) các phương thức final và private vì nó được thiết kế nhằm khuyến khích các nguyên tắc kiểm thử tốt và đề cao việc sử dụng các nguyên tắc thiết kế hướng đối tượng chính xác.

- **Phương thức final:** Mockito không hỗ trợ stubbing các phương thức final vì điều này vi phạm khái niệm về tính kiểm thử và tính linh hoạt. Phương thức final được thiết kế để không thể ghi đè, và bằng cách stubbing chúng, bạn sẽ bỏ qua hành vi ban đầu được định nghĩa bởi phương thức final. Điều này có thể dẫn đến kết quả không đoán trước được và làm suy yếu tính toàn vẹn của kiểm thử.
- **Phương thức private:** Mockito không hỗ trợ stubbing các phương thức private vì nó tuân theo nguyên tắc kiểm thử hành vi của một đối tượng thông qua giao diện công khai. Phương thức private là những chi tiết triển khai nội bộ không nên được kiểm thử hoặc stub. Bằng cách tập trung vào giao diện công khai, các kiểm thử trở nên mạnh mẽ hơn và chịu được sự thay đổi trong triển khai nội bộ của lớp.

Thay vì stubbing các phương thức final và private, Mockito khuyến khích các nhà phát triển thiết kế mã kiểm thử được kiểm thử bằng cách ưa chuộng việc tiêm phụ thuộc và tách biệt các vấn đề. Bằng cách sử dụng giao diện và các phương thức công khai được thiết kế đúng cách, bạn có thể tạo ra mã modul và dễ bảo trì hơn, dễ kiểm thử hơn.

Cần lưu ý rằng có các phương pháp và công cụ thay thế khác, chẳng hạn như PowerMock, cho phép stubbing các phương thức final và private. Tuy nhiên, việc sử dụng các công cụ như vậy nên được cân nhắc cẩn thận, vì nó có thể dẫn đến mã khó hiểu, khó bảo trì và khó tái cấu trúc. Thông thường, nên tránh việc stubbing các phương thức final và private và thay vào đó tập trung vào kiểm thử hành vi công khai của đối tượng của bạn.

Section 7: Powermock with Mockito

1. PowerMock

PowerMock là một framework mở rộng cho Mockito và EasyMock, cho phép bạn thực hiện các hoạt động mạnh mẽ hơn trong việc kiểm thử các mã có sử dụng các phương thức final, static, private, và các final class. Nó cung cấp các tính năng mở rộng để stub, mock và xác minh các phương thức và lớp mà các framework kiểm thử tiêu chuẩn không thể làm được.

PowerMock có thể được sử dụng để:

- Stub phương thức final: PowerMock cho phép bạn stub các phương thức final để kiểm thử các hành vi bên trong chúng.
- Stub phương thức static: PowerMock cho phép bạn stub và mock các phương thức static, giúp bạn kiểm thử các khối mã sử dụng các phương thức static.
- Stub và mock các phương thức private: PowerMock cho phép bạn stub và mock các phương thức private trong lớp, giúp bạn kiểm thử các phương thức và hành vi nội bộ không công khai.
- Xác minh việc gọi phương thức final, static và private: PowerMock cho phép bạn xác minh xem các phương thức final, static và private đã được gọi với đúng tham số và số lần gọi mong muốn.
- Mock các lớp cuối cùng (final classes): PowerMock cho phép bạn mock các lớp cuối cùng để kiểm thử các tương tác và kết quả trả về từ các lớp này.

Tuy nhiên, việc sử dụng PowerMock cần được cân nhắc cẩn thận vì nó có thể làm mã kiểm thử của bạn trở nên phức tạp và khó hiểu. Nếu có thể, hãy cân nhắc áp dụng thiết kế mã linh hoạt và tiêm phụ thuộc để làm cho mã kiểm thử dễ bảo trì và hiệu quả hơn. PowerMock thường được sử dụng trong các trường hợp đặc biệt khi không có sự lựa chọn khác để kiểm thử các phương thức final, static và private.

2. Using PowerMock and Mockito to mock a Static Method.

Khi sử dụng PowerMock và Mockito cùng nhau, bạn có thể mock các phương thức static bằng cách làm theo các bước sau:

Bước 1: Thêm các phụ thuộc cần thiết

Để sử dụng PowerMock và Mockito, bạn cần thêm các phụ thuộc cần thiết vào dự án của mình. Dưới đây là các phụ thuộc cho Maven:

```
<dependency>
    <groupId>org.powermock</groupId>
    <artifactId>powermock-api-mockito</artifactId>
    <version>1.6.4</version>
```

```
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.powermock</groupId>
    <artifactId>powermock-module-junit4</artifactId>
    <version>1.6.4</version>
    <scope>test</scope>
</dependency>
```

Bước 2: Chuẩn bị lớp kiểm thử

Đánh dấu lớp kiểm thử của bạn bằng chú thích @RunWith(PowerMockRunner.class) để chỉ định JUnit sử dụng PowerMock runner. Ngoài ra, đánh dấu lớp kiểm thử bằng @PrepareForTest(ClassWithStaticMethod.class), chỉ định lớp chứa phương thức static mà bạn muốn mock.

```
@RunWith(PowerMockRunner.class)
@PrepareForTest(ClassWithStaticMethod.class)
public class ExampleTest {
    // ...
}
```

// Code hữu ích khác

```
@RunWith(PowerMockRunner.class)
@PrepareForTest({ UtilityClass.class})

PowerMockito.mockStatic(UtilityClass.class);
when(UtilityClass.staticMethod(anyLong())).thenReturn(150);

PowerMockito.verifyStatic();
UtilityClass.staticMethod(1 + 2 + 3);
```

Bước 3: Mock phương thức static

Trong phương thức kiểm thử của bạn, bạn có thể mock phương thức static bằng cách sử dụng phương thức PowerMockito.mockStatic(). Sau đó, bạn có thể xác định hành vi mong muốn cho phương thức static bằng cách sử dụng stubbing thông thường của Mockito.

```
@Test
public void testStaticMethod() {
    PowerMockito.mockStatic(ClassWithStaticMethod.class);
    Mockito.when(ClassWithStaticMethod.staticMethod()).thenReturn("Giá trị
giả định");

    // Kiểm thử mã của bạn tương tác với phương thức static

    // Xác minh tương tác với phương thức static
    PowerMockito.verifyStatic(ClassWithStaticMethod.class);
    ClassWithStaticMethod.staticMethod();
}
```

Trong ví dụ này, ClassWithStaticMethod là lớp chứa phương thức static mà bạn muốn mock. Bằng cách sử dụng PowerMockito.mockStatic(), bạn chỉ định PowerMock mock phương thức static. Sau đó, bạn có thể xác định giá trị trả về mong muốn cho phương thức static bằng Mockito.when(). Cuối cùng, bạn có thể kiểm thử mã của bạn tương tác với phương thức static và xác minh tương tác bằng PowerMockito.verifyStatic().

Hãy chắc chắn xử lý đúng các ngoại lệ hoặc cài đặt bổ sung yêu cầu từ phương thức static đang được mock.

Bằng cách kết hợp PowerMock và Mockito, bạn có thể dễ dàng mock và kiểm thử các lớp sử dụng phương thức static, điều mà thông thường rất khó để mock chỉ với Mockito một mình.

Ví dụ:

```
@RunWith(PowerMockRunner.class)
@PrepareForTest({ UtilityClass.class})
public class PowerMockitoMockingStaticMethodTest {

    @Mock
    Dependency dependencyMock;

    @InjectMocks
    SystemUnderTest systemUnderTest;

    @Test
    public void powerMockito_MockingAStaticMethodCall() {

        when(dependencyMock.retrieveAllStats()).thenReturn(
              Arrays.asList(1, 2, 3));

        PowerMockito.mockStatic(UtilityClass.class);

        when(UtilityClass.staticMethod(anyLong())).thenReturn(150);

        assertEquals(150, systemUnderTest.methodCallingAStaticMethod());

        //To verify a specific method call
        //First : Call PowerMockito.verifyStatic()
        //Second : Call the method to be verified
        PowerMockito.verifyStatic();
        UtilityClass.staticMethod(1 + 2 + 3);

        // verify exact number of calls
        //PowerMockito.verifyStatic(Mockito.times(1));

    }
}
```

3. Using PowerMock and Mockito to invoke a private Method(Step16)

Trong ví dụ sau, chúng ta sẽ tạo một mock của các phương thức **private**.

Bước 1: Tạo một lớp chứa một private method. Chúng tôi đã tạo lớp có tên **Utility** và xác định một phương thức private và một phương thức public (trả về đối tượng private method).

```
public class Utility {

    private String privateMethod(String message) {
       return message;
    }

    public String callPrivateMethod(String message) {
       return privateMethod(message);
    }
}
```

Bước 2: Tạo một JUnit test case có tên **PowerMock_test**.

```
import static junit.framework.Assert.assertEquals;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.powermock.api.mockito.PowerMockito;
import org.powermock.core.classloader.annotations.PrepareForTest;
import org.powermock.modules.junit4.PowerMockRunner;

@RunWith(PowerMockRunner.class)
@PrepareForTest(Utility.class)
public class Powermock_test {

    @Test
    public void TestPrivateMethod_WithPowerMock() throws Exception {

       String message = " PowerMock with Mockito and JUnit ";
       String expectedmessage = " Using with EasyMock ";

       Utility mock =PowerMockito.spy(new Utility());
       PowerMockito.doReturn(expectedmessage).when(mock, "privateMethod",
message);

       String actualmessage = mock.callPrivateMethod(message);
       assertEquals(expectedmessage, actualmessage);

       System.out.println(PowerMockito.verifyPrivate(getClass()));
    }
}
```

4. Mocking a Constructor

Đôi khi, chúng ta cần kiểm tra một constructor của một lớp, bao gồm cả các đối số đã được truyền vào. Để giải quyết vấn đề này, chúng ta có thể sử dụng PowerMock để mock đối tượng và kiểm tra constructor.

Dưới đây là một ví dụ minh họa:

```
public class MyClass {
    private int value;

    public MyClass(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

@RunWith(PowerMockRunner.class)
@PrepareForTest(MyClass.class)
public class MyClassTest {
    @Test
    public void testConstructor() throws Exception {
        int expectedValue = 5;
        PowerMockito.whenNew(MyClass.class)
                .withArguments(expectedValue)
                .thenReturn(PowerMockito.mock(MyClass.class));

        MyClass myClass = new MyClass(expectedValue);

        assertEquals(expectedValue, myClass.getValue());
    }
}
```

Trong ví dụ này, ta sử dụng PowerMockito để mock đối tượng MyClass và kiểm tra constructor đã được gọi với đúng đối số hay không. Để làm điều này, ta sử dụng phương thức whenNew() để mock constructor và withArguments() để truyền đối số cho constructor được gọi. Sau đó, ta kiểm tra xem giá trị trả về của constructor có đúng như mong đợi không.

Tuy nhiên, cần lưu ý rằng việc sử dụng PowerMock để mock constructor có thể làm giảm tính đóng gói của mã nguồn và không nên được sử dụng quá nhiều.