MỤC TIÊU:

Kết thúc bài thực hành này bạn có khả năng

✓ Hiểu được các khái niệm **Test Fixture**,tổ chức cấu trúc test case với test fixture

✓ Hiểu được các khái niệm Test Suites, tạo và thực thi test suites ✓ Vận dụng linh hoạt các annotation, class trong junit

YÊU CẦU:

✓ **Viết testcase -> Kiểm thử với Junit**

✓ **Thiết kế testcase theo đúng format đã hướng dẫn, rõ ràng khoa học.**

PHẦN I

Bài 1 (1.0 điểm)

Sinh viên thực hiện xây dựng 1 chương trình kiểm thử ứng dụng có 2 phương thức là tính giai thừa một số nguyên không âm và tính tổng 2 số nguyên, có sử dụng test fixture

➢ Tạo class có tên MathFunc, xây dưng 2 phương thức factorial và plus:

LAB5 : KIỂM THỬ

➢ Tạo junit test case, tạo class có tên MathFuncTest chứa các test case cần thiết để test 2 phương thức bên trên.

➢ Sử dụng " @Before và @After " quy định phương thức chạy trước và sau các test case.

```
@Before
public void init() { math = new MathFunc(); }
@After
public void tearDown() { math = null; }
```

➢ Dùng annotation @Ignore để bỏ qua một test case

```
@Ignore
@Test
public void todo() {
    assertTrue(math.plus(1, 1) == 3);
}
```

➢ Class MathFuncTest được code như sau:

```
public class MathFuncTest {
    private MathFunc math;

    @Before
    public void init() { math = new MathFunc(); }
    @After
    public void tearDown() { math = null; }

    @Test
    public void calls() {
        assertEquals(0, math.getCalls());

        math.factorial(1);
        assertEquals(1, math.getCalls());

        math.factorial(1);
        assertEquals(2, math.getCalls());
    }
```

```
    @Test
    public void factorial() {
        assertTrue(math.factorial(0) == 1);
        assertTrue(math.factorial(1) == 1);
        assertTrue(math.factorial(5) == 120);
    }

    @Test(expected = IllegalArgumentException.class)
    public void factorialNegative() {
        math.factorial(-1);
    }

    @Ignore
    @Test
    public void todo() {
        assertTrue(math.plus(1, 1) == 3);
    }
}
```

➢ Sinh viên chú ý :

- Phương thức public void calls() kiểm tra so sanh giá trị của biến "calls"
- Phương thức public void factorial() kiểm tra phương thức tính giai thừa
- Phương thức factorialNegative() kiểm tra trường hợp số âm
- Phương thức public void todo() kiểm tra phướng thức tính tổng

➢ Tạo class thực thi có chưa main như sau:

```
public static void main(String[] args) throws Exception {
    JUnitCore runner = new JUnitCore();
    Result result = runner.run(MathFuncTest.class);
    System.out.println("run tests: " + result.getRunCount());
    System.out.println("failed tests: " + result.getFailureCount());
    System.out.println("ignored tests: " + result.getIgnoreCount());
    System.out.println("success: " + result.wasSuccessful());
}
```

➢ Chạy kiểm tra kết quả:

```
run tests: 3
failed tests: 0
ignored tests: 1
success: true
```

Bài 2 (1.0 điểm)

Tạo JUnit Test Suite sử dụng @RunWith @Suite. Giả sử có 2 lớp SuiteTest1.java và SuiteTest2.java, mỗi class chứa những test case khác nhau.

➢ Tạo class SuiteTest1.java với mục đích xuất ra một chuỗi ký tự

```
package fpoly.junit;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class SuiteTest1 {

    public String message = "Fpoly";


    JUnitMessage junitMessage = new JUnitMessage(message);


    @Test(expected = ArithmeticException.class)

    public void testJUnitMessage() {

        System.out.println("Junit Message is printing ");

        junitMessage.printMessage();

    }
```

```
    @Test
    public void testJUnitHiMessage() {
        message = "Hi!" + message;

        System.out.println("Junit Hi Message is printing ");

        assertEquals(message, junitMessage.printHiMessage());

        System.out.println("Suite Test 2 is successful " + message);

    }
}
```

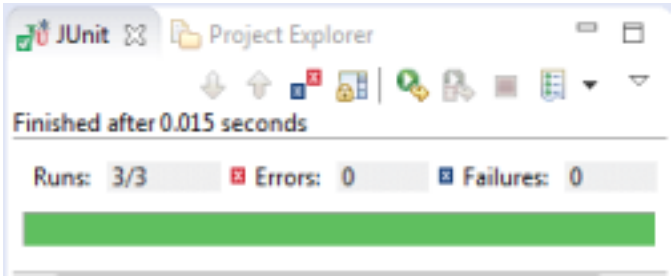
➢ Tương tự tạo class SuiteTest2.java

```
package fpoly.junit;

import org.junit.Assert;
import org.junit.Test;

public class SuiteTest2 {


    @Test
    public void createAndSetName() {


        String expected = "Y";
        String actual = "Y";

        Assert.assertEquals(expected, actual);

        System.out.println("Suite Test 1 is successful " + actual);

    }

}
```

➢ Tạo class JunitTest.java để thực thi 2 lớp Suitest bên trên


```
package fpoly.junit;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
  SuiteTest1.class,
  SuiteTest2.class,
})

public class JunitTest {
                        // This class remains empty, it is used only as a hold
er for the above annotations
}
```

Chạy test và kiểm tra kết quả



Bài 3 (1.0 điểm)

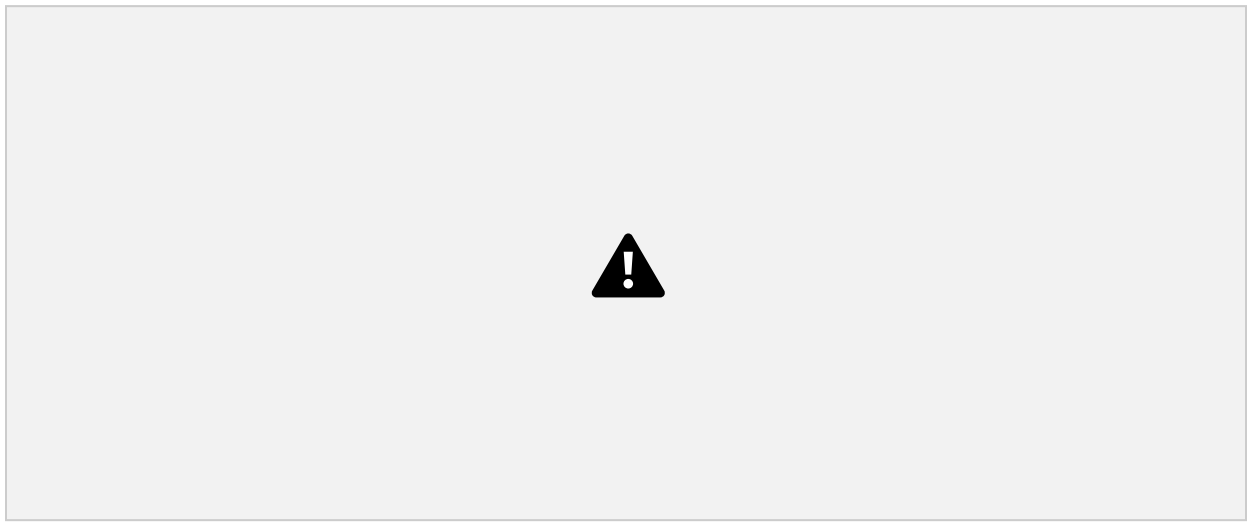
Xây dựng lớp kiểm thử với JUnit Annotations: @Before , @BeforeClass, @After, @AfterClass, @Test, @Ignore…

➢ Tạo lớp kiểm thử JunitAnnotationsExample.java

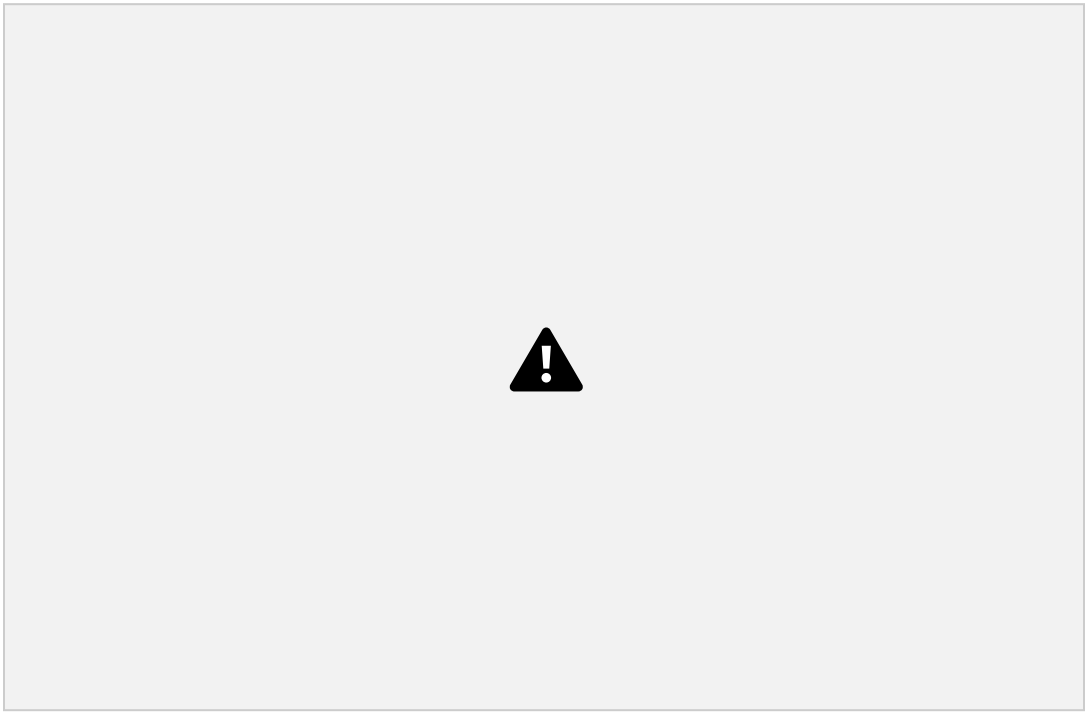
```
package fpoly.junit;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;

import java.util.ArrayList;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;

public class JunitAnnotationsExample {

    private ArrayList<String> list;

    @BeforeClass
    public static void m1() {
        System.out.println("Using @BeforeClass , executed before all test cases ");

    }
```

```
    @Before
    public void m2() {
        list = new ArrayList<String>();
        System.out.println("Using @Before annotations ,executed before each test cases ");

    }
```

```
@AfterClass
public static void m3() {
    System.out.println("Using @AfterClass ,executed after all test cases");

}

@After
public void m4() {
    list.clear();
    System.out.println("Using @After ,executed after each test cases");

}

@Test
public void m5() {
    list.add("test");
    assertFalse(list.isEmpty());
    assertEquals(1, list.size());
}

@Ignore
public void m6() {
    System.out.println("Using @Ignore , this execution is ignored");

}
```

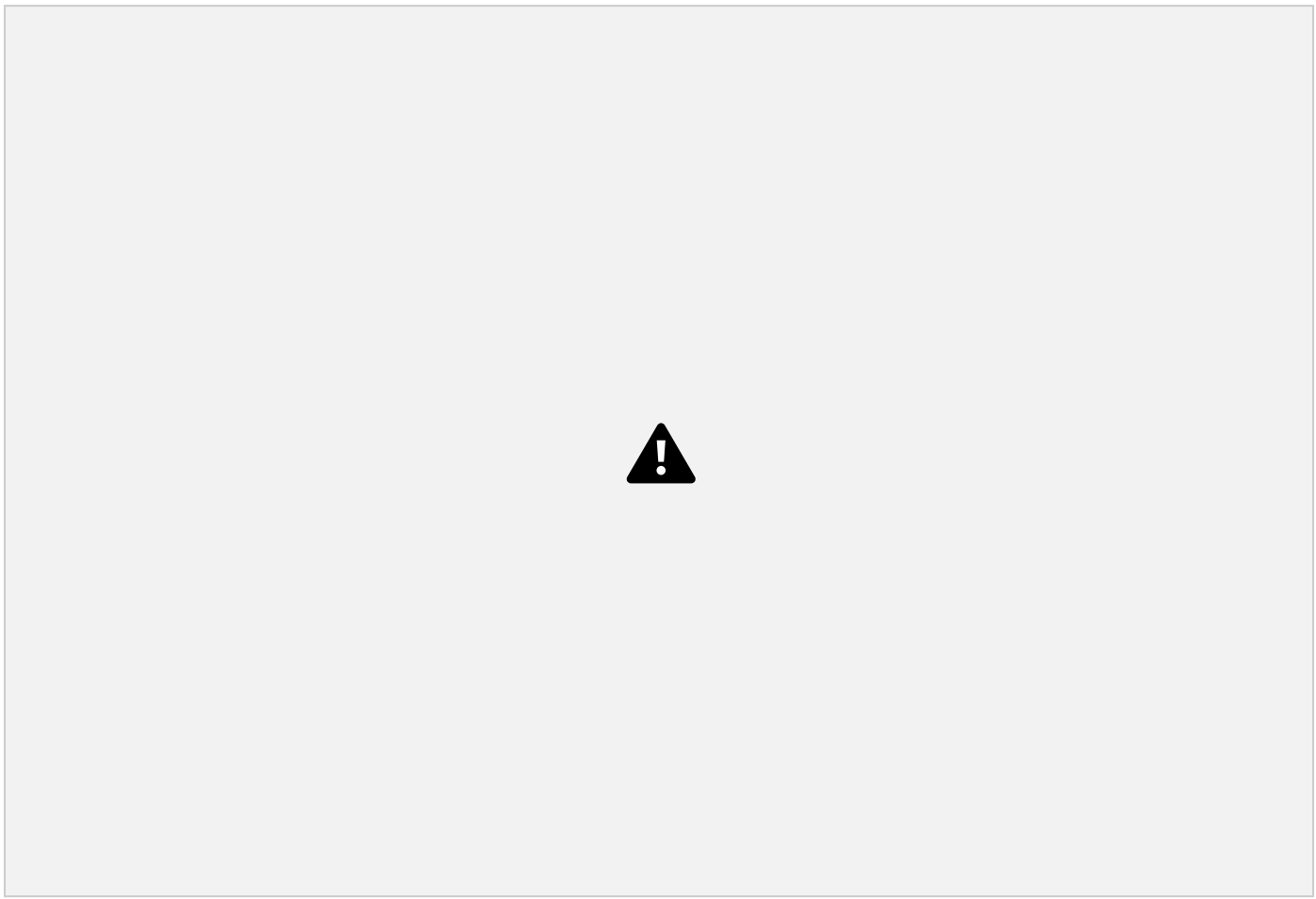


➢ Tạo lớp TestRunner.java thực thi đoạn test case bên trên



Chạy và kiểm tra kết quả

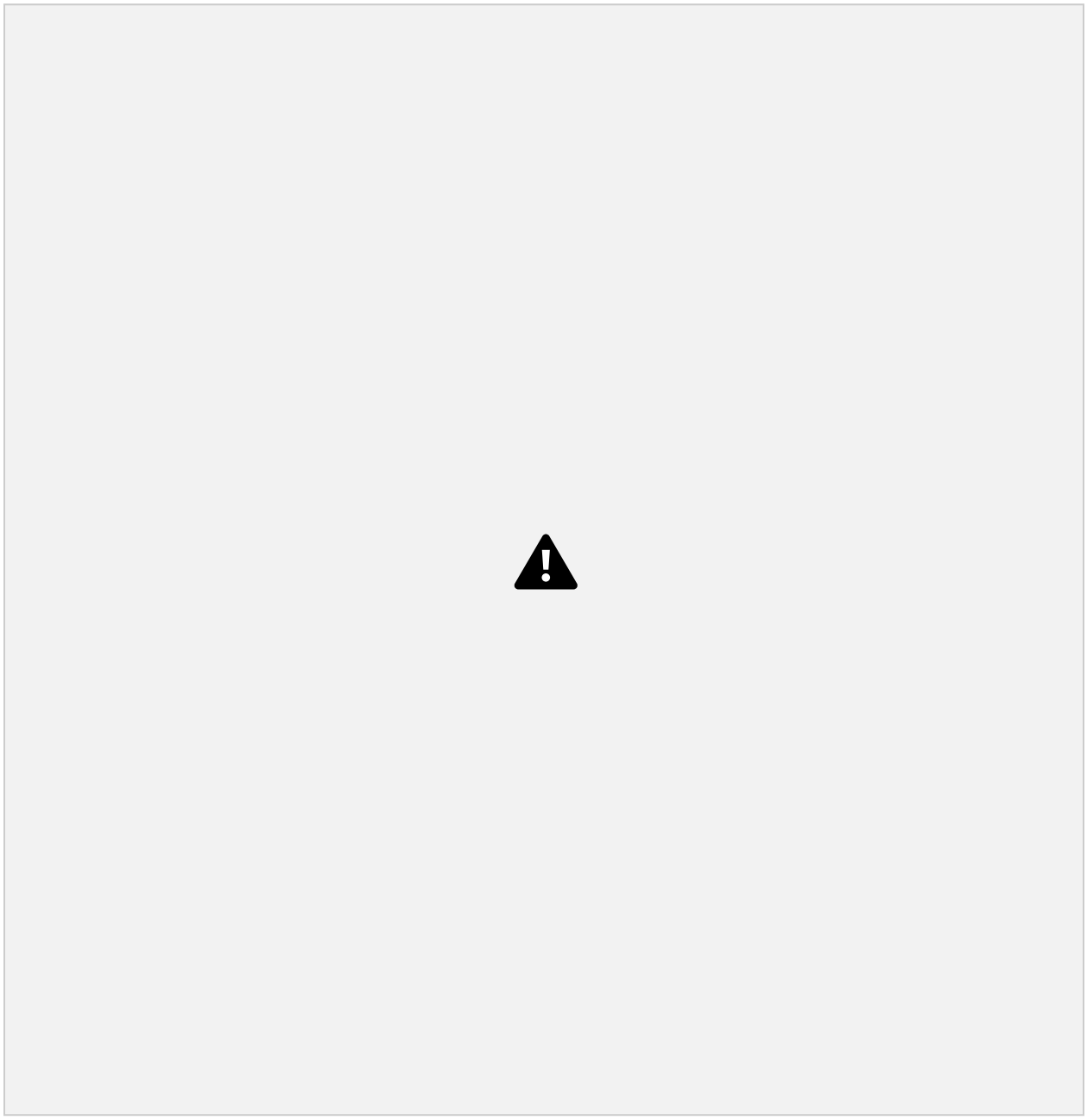
Bài 4. (3.0. điểm). Đặc tả yêu cầu: Cách tính tiền căn cứ vào lứa tuổi như sau:



Tạo form, thiết kế testcase, kiểm thử cho các trường hợp trên.

Gợi ý: cho phép sử dụng các phương pháp kết hợp.

Bài 5. (4.0 điểm).



Câu 1. Tạo Form Đăng Ký Tài Khoản Khách Hàng như giao diện trên:

Yêu cầu chi tiết của form:

1. Mã Khách Hàng:
   - Là trường bắt buộc (Required).
   - Độ dài tfi 6 đến 10 ký tự.
   - Chỉ cho phép nhập chfi cái (a-z, A-Z) và số (0-9).
   - Mã khách hàng phải là duy nhất (không được trùng lặp).

2. Họ và Tên:

   - Là trường bắt buộc.

• Độ dài tfi 5 đến 50 ký tự, cho phép nhập tiếng Việt có dấu và khoảng trắng.

3. Email:

• Là trường bắt buộc.

• Phải có định dạng email hợp lệ (ví dụ: nguyenvana@email.com). •

Email không được trùng lặp.

4. Số điện thoại:

• Là trường bắt buộc.

• Chỉ cho phép nhập số (0-9).

• Độ dài tfi 10 đến 12 ký tự và phải bắt đầu bằng số 0.

5. Địa chỉ:

• Là trường bắt buộc.

• Độ dài tối đa 255 ký tự.

6. Mật khẩu:

• Là trường bắt buộc.

• Độ dài tối thiểu 8 ký tự.

7. Xác nhận Mật khẩu:

• Là trường bắt buộc.

• Phải khớp chính xác với trường Mật khẩu.

8. Ngày sinh:

• Là trường không bắt buộc.

• Nếu nhập, người dùng phải đủ 18 tuổi (tính đến ngày hiện tại).

9. Giới tính:

• Là trường không bắt buộc.

10. Điều khoản dịch vụ:

• Bắt buộc phải được tích chọn. CÁC NÚT

CHỨC NĂNG:

• Đăng ký: Khi nhấn, kiểm tra (validate) toàn bộ dfi liệu. Nếu tất cả đều hợp lệ, hiển thị thông báo "Đăng ký tài khoản thành công!".

• Nhập lại: Khi nhấn, xóa toàn bộ dfi liệu đã nhập trên form, trả các trường về trạng thái mặc định.

THÔNG BÁO LỖI:

• Hiển thị các thông báo lỗi cụ thể, rõ ràng nếu người dùng nhập thông tin không hợp lệ cho bất kỳ trường nào.

Câu 2:

Viết các test case (dưới dạng bảng) để kiểm thử chức năng của form Đăng Ký Tài Khoản Khách Hàng dựa trên tất cả các ràng buộc và yêu cầu đã được mô tả trong Câu 1.

Hết

Good luck!