# گزارش تکلیف اول درس مباحث ویژه در نرم افزار 1

نسترن عشوری – 810101225

کد ها ضمیمه شده است. همچنین توضیحات و کد ها در این لینک قابل مشاهده است.

سوال اول )

الف –

- کلاس Play

این کلاس شامل نام نمایش و ژانر آن نمایش است.

برای تست کردن، یک نمونه از کلاس با مقادیر اولیه ای می سازیم و سپس تست میکنیم آیا مقادیر متغیرهای آن کلاس با مقادیر داده شده برابر است یا نه.

```
@Test
public void testPlay() {
    Play play = new Play(name:"Hamlet", type:"tragedy");
    assertEquals("Hamlet", play.name);
    assertEquals("tragedy", play.type);
}
```

میتوانیم تست کنیم که اگر رشته خالی یا null به این کلاس بدهیم چه اتفاقی رخ میدهد.

```java
@Test
public void testPlayWithEmptyStrings() {
    Play play = new Play(name:"", type:"");
    assertEquals("", play.name);
    assertEquals("", play.type);
}

@Test
public void testPlayWithNullValues() {
    Play play = new Play(name:null, type:null);
    assertNull(play.name);
    assertNull(play.type);
}
```

میبینیم که تست ها pass میشود. ولی این کار به نظر منطقی نیست. باید جلوی ست شدن متغیرهای کلاس با مقادیر null را گرفت. پس کلاس را به شکل زیر تغییر میدهیم.

```java
package src.main.java.dramaplays.model;

...
public class Play {

    public String name;
    public String type;

    public Play(String name, String type) {
        if (name == null || name.trim().isEmpty()) {
            throw new IllegalArgumentException(s:"Name cannot be null or empty");
        }
        if (type == null || type.trim().isEmpty()) {
            throw new IllegalArgumentException(s:"Type cannot be null or empty");
        }
        this.name = name;
        this.type = type;
    }
}
```

حال مجموعه تست ها را به شکل زیر آپدیت میکنیم.

```java
@Test
public void testPlayConstructorWithNullName() {
    Exception exception = assertThrows(IllegalArgumentException.class, () -> {
        new Play(name:null, type:"Tragedy");
    });

    String expectedMessage = "Name cannot be null or empty";
    String actualMessage = exception.getMessage();

    assertTrue(actualMessage.contains(expectedMessage));
}

@Test
public void testPlayConstructorWithEmptyName() {
    Exception exception = assertThrows(IllegalArgumentException.class, () -> {
        new Play(name:"", type:"Tragedy");
    });

    String expectedMessage = "Name cannot be null or empty";
    String actualMessage = exception.getMessage();

    assertTrue(actualMessage.contains(expectedMessage));
}
```

```java
@Test
public void testPlayConstructorWithNullType() {
    Exception exception = assertThrows(IllegalArgumentException.class, () -> {
        new Play(name:"Hamlet", type:null);
    });

    String expectedMessage = "Type cannot be null or empty";
    String actualMessage = exception.getMessage();

    assertTrue(actualMessage.contains(expectedMessage));
}

@Test
public void testPlayConstructorWithEmptyType() {
    Exception exception = assertThrows(IllegalArgumentException.class, () -> {
        new Play(name:"Hamlet", type:"");
    });

    String expectedMessage = "Type cannot be null or empty";
    String actualMessage = exception.getMessage();

    assertTrue(actualMessage.contains(expectedMessage));
}
```

- کلاس Performance

این کلاس شامل آیدی نمایش و همچنین تعداد شرکت کنندگان آن اجرا است.

برای تست کردن، یک نمونه از کلاس با مقادیر اولیه ای می سازیم و سپس تست میکنیم آیا مقادیر متغیرهای آن کلاس با مقادیر داده شده برابر است یا نه.

```
@Test
public void testPerformance() {
    Performance performance = new Performance(playID:"1", audience:200);
    assertEquals("1", performance.playID);
    assertEquals(200, performance.audience);
}
```

میدانیم که آیدی نباید null باشد. همچنین تعداد شرکت کننده ها نباید منفی باشد. بعد از ساخت شی آیدی نباید تغییر کند اما تعداد شرکت کننده را میشود عوض کرد.

پس کلاس را به صورت زیر تغییر میدهیم.

```java
package src.main.java.dramaplays.model;

import java.util.HashSet;
import java.util.Set;

public class Performance {

    public final String playID;
    public int audience;

    public Performance(String playID, int audience) {
        if (playID == null || playID.trim().isEmpty()) {
            throw new IllegalArgumentException(s:"Play ID cannot be null or empty");
        }
        if (audience < 0) {
            throw new IllegalArgumentException(s:"Audience cannot be negative");
        }
        this.playID = playID;
        this.audience = audience;
    }

    public String getPlayID() {
        return playID;
    }

    public int getAudience() {
        return audience;
    }

    public void setAudience(int audience) {
        if (audience < 0) {
            throw new IllegalArgumentException(s:"Audience cannot be negative");
        }
        this.audience = audience;
    }
}
```

حال برای هر یک از ویژگیهای ذکر شده تست مینویسیم.

```java
@Test
public void testPerformanceConstructorWithValidInputs() {
    Performance performance = new Performance(playID:"play1", audience:100);
    assertNotNull(performance);
    assertEquals("play1", performance.getPlayID());
    assertEquals(100, performance.getAudience());
}

@Test
public void testPerformanceConstructorWithNullID() {
    Exception exception = assertThrows(IllegalArgumentException.class, () -> {
        new Performance(playID:null, audience:100);
    });

    String expectedMessage = "Play ID cannot be null or empty";
    String actualMessage = exception.getMessage();

    assertTrue(actualMessage.contains(expectedMessage));
}

@Test
public void testPerformanceConstructorWithEmptyID() {
    Exception exception = assertThrows(IllegalArgumentException.class, () -> {
        new Performance(playID:"", audience:100);
    });

    String expectedMessage = "Play ID cannot be null or empty";
    String actualMessage = exception.getMessage();

    assertTrue(actualMessage.contains(expectedMessage));
}
```

```java
@Test
public void testPerformanceConstructorWithNegativeAudience() {
    Exception exception = assertThrows(IllegalArgumentException.class, () -> {
        new Performance(playID:"play1", -1);
    });

    String expectedMessage = "Audience cannot be negative";
    String actualMessage = exception.getMessage();

    assertTrue(actualMessage.contains(expectedMessage));
}

@Test
public void testSetAudienceWithNegativeValue() {
    Performance performance = new Performance(playID:"play2", audience:100);

    Exception exception = assertThrows(IllegalArgumentException.class, () -> {
        performance.setAudience(-1);
    });

    String expectedMessage = "Audience cannot be negative";
    String actualMessage = exception.getMessage();

    assertTrue(actualMessage.contains(expectedMessage));
}
```

- کلاس Invoice

این کلاس شامل نام مشتری و همچنین لیستی از نمایش هاست. هنگام صدا شدن تابع سازنده این کلاس، این دو متغیر مقداردهی میشوند.

برای تست کردن، یک نمونه از کلاس با مقادیر اولیه ای می سازیم و سپس تست میکنیم آیا مقادیر متغیرهای آن کلاس با مقادیر داده شده برابر است یا نه.

همچنین باید تست شود customer برابر null نباشد. اما performances میتواند null باشد.

پس تابع را به صورت زیر تغییر میدهیم.

```java
package src.main.java.dramaplays.model;

import java.util.List;

...
public class Invoice {

    public String customer;
    public List<Performance> performances;

    public Invoice(String customer, List<Performance> performances) {
        if (customer == null || customer.trim().isEmpty()) {
            throw new IllegalArgumentException(s:"Customer cannot be null or empty");
        }
        this.customer = customer;
        this.performances = performances == null ? List.of() : List.copyOf(performances);
    }

    public String getCustomer() {
        return customer;
    }

    public List<Performance> getPerformances() {
        return List.copyOf(performances); // Return an immutable copy
    }
}
```

حال با مجموعه تست زیر کلاس را تست میکنیم.

```java
@Test
public void testInvoiceConstructorWithValidInputs() {
    Performance performance1 = new Performance(playID:"play1", audience:100);
    Performance performance2 = new Performance(playID:"play2", audience:200);
    List<Performance> performances = List.of(performance1, performance2);

    Invoice invoice = new Invoice(customer:"Customer1", performances);
    assertNotNull(invoice);
    assertEquals("Customer1", invoice.getCustomer());
    assertEquals(2, invoice.getPerformances().size());
    assertTrue(invoice.getPerformances().contains(performance1));
    assertTrue(invoice.getPerformances().contains(performance2));
}

@Test
public void testInvoiceConstructorWithNullCustomer() {
    Performance performance = new Performance(playID:"play1", audience:100);
    List<Performance> performances = List.of(performance);

    Exception exception = assertThrows(IllegalArgumentException.class, () -> {
        new Invoice(customer:null, performances);
    });

    String expectedMessage = "Customer cannot be null or empty";
    String actualMessage = exception.getMessage();

    assertTrue(actualMessage.contains(expectedMessage));
}

@Test
public void testInvoiceConstructorWithEmptyCustomer() {
    Performance performance = new Performance(playID:"play1", audience:100);
    List<Performance> performances = List.of(performance);

    Exception exception = assertThrows(IllegalArgumentException.class, () -> {
        new Invoice(customer:"", performances);
    });

    String expectedMessage = "Customer cannot be null or empty";
    String actualMessage = exception.getMessage();

    assertTrue(actualMessage.contains(expectedMessage));
}
```

با تغییرات ایجاد شده همه تست ها pass میشوند.

ب –

توضیح تابع:

کلاس `FactorPrinter` برای چاپ صورتحساب (فاکتور) مربوط به اجراهای تئاتر یک مشتری استفاده می‌شود. این کلاس شامل یک متد به نام `print` است که دو ورودی می‌گیرد: یک شیء از کلاس `Invoice` که اطلاعات مشتری و اجرای نمایش‌ها را نگه می‌دارد، و یک نقشه (`Map`) که اطلاعات نمایش‌ها را بر اساس شناسه آن‌ها ذخیره می‌کند. این متد محاسبات مربوط به هزینه هر نمایش و اعتباراتی که مشتری به دست آورده را انجام می‌دهد و سپس خروجی نهایی را به صورت یک رشته قالب‌بندی شده بازمی‌گرداند.

در این متد، بسته به نوع نمایش (تراژدی یا کمدی)، هزینه‌های مختلفی محاسبه می‌شود. برای نمایش‌های تراژدی، اگر تعداد تماشاگران بیش از ۳۰ نفر باشد، هزینه اضافی برای هر تماشاگر محاسبه می‌شود. در مورد نمایش‌های کمدی، علاوه بر هزینه اولیه، هزینه اضافی برای تعداد تماشاگران بیش از ۲۰ نفر و همچنین هزینه اضافی برای هر تماشاگر محاسبه می‌شود. همچنین، اعتباراتی بر اساس تعداد تماشاگران و نوع نمایش به دست می‌آید. در نهایت، متد `print` یک رشته شامل جزئیات فاکتور، مبلغ کل بدهی و اعتبارهای کسب شده را به صورت قالب‌بندی شده برمی‌گرداند.

سناریوهای زیر تست شده است.

1. یک اجرای تراژدی با کمتر از 30 تماشاگر
2. یک اجرای تراژدی با بیشتر از 30 تماشاگر
3. یک اجرای کمدی با کمتر از 20 تماشاگر
4. یک اجرای کمدی با بیش از 20 تماشاگر
5. یک فاکتور با چند اجرا
6. نمایش با تایپ unknown
7. نمایش تراژدی با صفر تماشاگر
8. نمایش کمدی با صفر تماشاگر
9. چند نمایش تراژدی
10. چند نمایش کمدی

مشاهده میکنیم که همه تست ها pass میشوند. در کد جزییات تست ها آورده شده است.

سوال دوم )

تست های زیر طراحی شده و همه pass شده اند. جزییات در کد قابل مشاهده است.

testHasPassedPre_AllPrePassed:

در این تست، دو درس پیش‌نیاز ۱۰۱ و ۱۰۲ با نمرات ۱۰ و ۱۲ گذرانده شده‌اند.

تابع باید true بازگرداند زیرا تمام پیش‌نیازها پاس شده‌اند.

testHasPassedPre_OnePreNotPassed:

در این تست، درس ۱۰۱ با نمره ۱۰ و درس ۱۰۲ با نمره ۹ گذرانده شده‌اند.

تابع باید false بازگرداند زیرا یکی از پیش‌نیازها پاس نشده است.

testHasPassedPre_AllPrePassedWithMehman:

در این تست، دو درس پیش‌نیاز ۱۰۱ و ۱۰۲ با نمرات ۱۲ و ۱۳ گذرانده شده‌اند و هر دو دانشجوی مهمان (مه‌مان) هستند.

تابع باید true بازگرداند زیرا تمام پیش‌نیازها با نمرات بالای ۱۲ پاس شده‌اند.

testHasPassedPre_OnePreNotPassedWithMehman:

در این تست، درس ۱۰۱ با نمره ۱۲ و درس ۱۰۲ با نمره ۱۱ گذرانده شده‌اند و هر دو دانشجوی مهمان هستند.

تابع باید false بازگرداند زیرا یکی از پیش‌نیازها با نمره زیر ۱۲ پاس نشده است.

testHasPassedPre_NoPreRequired:

در این تست، درسی بدون هیچ پیش‌نیاز بررسی می‌شود.

تابع باید true بازگرداند زیرا هیچ پیش‌نیازی وجود ندارد که پاس نشده باشد.

testHasPassedPre_NoRecords:

در این تست، هیچ رکوردی از نمرات وجود ندارد و درس پیش‌نیازهایی دارد.

تابع باید false بازگرداند زیرا هیچ پیش‌نیازی پاس نشده است.

testHasPassedPre_EmptyPrerequisites:

در این تست، درس پیش‌نیاز ندارد.

تابع باید true بازگرداند زیرا هیچ پیش‌نیازی وجود ندارد.

testHasPassedPre_AllPreFailed:

در این تست، هر دو درس پیش‌نیاز ۱۰۱ و ۱۰۲ با نمرات زیر ۱۰ (۸ و ۷) گذرانده شده‌اند.

تابع باید false بازگرداند زیرا هیچ یک از پیش‌نیازها پاس نشده‌اند.

testHasPassedPre_MultipleTerms:

در این تست، درس‌های پیش‌نیاز ۱۰۱ و ۱۰۲ در ترم‌های مختلف با نمرات ۱۰ و ۱۲ گذرانده شده‌اند.

تابع باید true بازگرداند زیرا تمام پیش‌نیازها پاس شده‌اند.

testHasPassedPre_SinglePrePassed:

در این تست، تنها یک درس پیش‌نیاز (۱۰۱) با نمره ۱۰ گذرانده شده است.

تابع باید true بازگرداند زیرا پیش‌نیاز پاس شده است.

testHasPassedPre_SinglePreFailed:

در این تست، تنها یک درس پیش‌نیاز (۱۰۱) با نمره ۸ گذرانده شده است.

تابع باید false بازگرداند زیرا پیش‌نیاز پاس نشده است.

testHasPassedPre_ExactBoundaryNonMehman:

در این تست، درس پیش‌نیاز (۱۰۱) با نمره دقیقاً ۱۰ گذرانده شده است و دانشجو مه‌مان نیست.

تابع باید true بازگرداند زیرا نمره دقیقاً برابر با حداقل نمره قبولی است.

testHasPassedPre_ExactBoundaryMehman:

در این تست، درس پیش‌نیاز (۱۰۱) با نمره ۱۲ گذرانده شده است و دانشجو مه‌مان است.

تابع باید true بازگرداند زیرا نمره دقیقاً برابر با حداقل نمره قبولی برای دانشجوی مه‌مان است.

testHasPassedPre_BelowBoundaryMehman:

در این تست، درس پیش‌نیاز (۱۰۱) با نمره ۱۱ گذرانده شده است و دانشجو مه‌مان است.

تابع باید false بازگرداند زیرا نمره کمتر از حداقل نمره قبولی برای دانشجوی مه‌مان است.

testHasPassedPre_MultipleRecordsSameCourse:

در این تست، درس پیش‌نیاز (۱۰۱) در دو ترم مختلف با نمرات ۸ و ۱۰ گذرانده شده است.

تابع باید true بازگرداند زیرا نمره بالاتر (۱۰) پاس شده است و دانشجو درس را در نهایت پاس کرده است.

سوال سوم )

الف)

عدم وجود دستور import برای کلاس مورد نظر :اگر SomeClass و متد aMethod در پکیج های جدا باشند باید آن‌ها را import کنیم.

عدم وجود assertion برای بررسی نتیجه :در یک تست واحد، باید از assertions برای بررسی نتیجه استفاده شود تا مطمئن شویم که نتیجه مورد انتظار حاصل شده است.

کد اصلاح شده :

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class SomeClassTest {

    @Test
    public void testA() {
        Integer result = new SomeClass().aMethod();
        System.out.println("Expected result is 10. Actual result is " + result);
        assertEquals(Integer.valueOf(10), result);
    }
}
```

ب)

استفاده نادرست از:annotation @Test

برای اعلام اینکه یک تست انتظار وقوع یک استثنا را دارد، باید از ویژگی expected در annotation @Testاستفاده شود. در کد فعلی، expects Exception یک ویژگی معتبر برای annotation @Test نیست.

نامشخص بودن نوع استثنا:

باید مشخص کنیم که کدام نوع استثنا انتظار می‌رود. به عنوان مثال، اگر انتظار داریم IllegalArgumentExceptionارخ دهد، باید این نوع را مشخص کنیم.

کد اصلاح شده:

```java
import org.junit.Test;

public class AnotherClassTest {

    @Test(expected = IllegalArgumentException.class)
    public void testProcessWithBadInput() {
        int badInput = 0;
        new AnotherClass().process(badInput);
    }
}
```

ج)

استفاده از متدهای initialize در تست‌ها:

فراخوانی متدهای initialize در هر تست به طور مجزا می‌تواند منجر به مشکلاتی شود زیرا وضعیت سیستم ممکن است ناپایدار یا ناهمگام شود.

عدم تضمین ترتیب اجرای تست‌ها:

ترتیب اجرای تست‌ها در JUnit تضمین نمی‌شود. بنابراین ممکن است testResourceAvailability قبل از testInitialization اجرا شود و در نتیجه منابع مورد نیاز آماده نباشند.

عدم بررسی نتیجه‌ی متد :initialize

هیچ بررسی‌ای برای اطمینان از موفقیت‌آمیز بودن اجرای متد initializeوجود ندارد.

برای رفع این مشکلات می‌توانیم از annotation های @Before یا @BeforeClassبرای اطمینان از مقداردهی اولیه‌ی منابع مورد نیاز قبل از اجرای تست‌ها استفاده کنیم. همچنین بهتر است بررسی کنیم که متدهای initialize به درستی اجرا شده‌اند.

توضیحات:

@BeforeClass برای اطمینان از اجرای متدهای initialize یک بار قبل از اجرای همه‌ی تست‌ها استفاده شده است. این تضمین می‌کند که منابع مورد نیاز برای همه تست‌ها آماده هستند.

متد testInitialization دیگر نیازی به فراخوانی متدهای initialize ندارد زیرا این کار در setUpBeforeClass انجام میشود.

متد testResourceAvailability همانطور که بود باقی مانده است اما اطمینان حاصل می‌شود که منابع قبل از اجرای تست آماده هستند.

کد اصلاح شده:

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

import static org.junit.Assert.assertTrue;

public class ResourceManagerTest {

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        Configuration.initialize();
        ResourceManager.initialize();
    }

    @Test
    public void testInitialization() {
    }

    @Test
    public void testResourceAvailability() {
        boolean isResourceAvailable = ResourceManager.isResourceAvailable("exampleResource");
        assertTrue(isResourceAvailable);
    }
}
```

سوال چهارم )

آزمون واحد (Unit Testing) یکی از روش‌های اصلی برای اطمینان از درستی عملکرد کد در برنامه‌نویسی است، اما وقتی صحبت از برنامه‌های چند ریسمانی (Multithreaded) می‌شود، موضوع پیچیده‌تر میشود. Race Conditions زمانی رخ می‌دهد که چندین ترد به طور همزمان به یک منبع مشترک دسترسی داشته باشند و ترتیب دسترسی به این منبع می‌تواند نتیجه را تغییر دهد. شناسایی و تست این شرایط با آزمون واحد بسیار دشوار است.

Thread Scheduling توسط سیستم‌عامل کنترل می‌شود و می‌تواند غیرقابل پیش‌بینی باشد. این موضوع باعث می‌شود که بازتولید دقیق مشکلات مربوط به زمان‌بندی در محیط تست واحد بسیار سخت باشد .قابلیت تکرارپذیری تست‌های واحد نیز چالشی دیگر است، به این معنا که با اجرای چندباره یک تست واحد باید نتایج مشابهی به دست آید، اما در برنامه‌های چند ریسمانی، عوامل خارجی مانند زمان‌بندی ریسمان‌ها و زمان‌های تاخیر می‌توانند باعث شوند که نتایج هر بار متفاوت باشد.

برای اطمینان از درستی برنامه‌های چند ریسمانی، علاوه بر آزمون واحد، می‌توان از آزمون‌های همزمانی (Concurrency Testing) استفاده کرد که به طور خاص برای شناسایی مشکلات مربوط به همزمانی طراحی شده‌اند و می‌توانند شامل تست‌های استرس (Stress Tests) و آزمون‌های بار (Load Tests) باشند .

استفاده از تکنیک‌های طراحی مناسب مانند عدم اشتراک‌گذاری داده‌های mutable بین ریسمان‌ها می‌تواند به کاهش احتمال بروز مشکلات کمک کند.