Software Testing CA#1

Student Name:
Pouya Haji Mohammadi Gohari

SID:810102113

Date of deadline
Thursday 16th May, 2024

Dept. of Computer Engineering

University of Tehran

Contents

1 Question 1 3
  1.1 Create tests for constructors 3
  1.2 Factor Printer Class 7

2 Question 2 13

3 Question 3 18

4 Question 4 20

1 Question 1

Firstly, we are going to clone from Question 1.Then we will get a copy of 4 classes from dramaplyas folder and create a gradle project named question 1.

1.1 Create tests for constructors

The first code we are going to implement tests for is ”Play”, where you can see the code in following structure:

```java
    package dramaplays;

    public class Play {

        public String name;
        public String type;

        public Play(String name, String type) {
            this.name = name;
            this.type = type;
        }
    }
```

To ensure the functionality of the Play class, we need to create test cases. While verification methods are necessary for confirming the correctness of our code, test cases help us build confidence in its proper functioning. By writing these test cases, we aim to validate that the Play class behaves as expected under various conditions.
Test cases scenarios:

1. When we create an instance of the ”Play” object, are the parameters set correctly?
2. If two instances of the ”Play” class are created with the same parameters, will their attributes be equal?
3. What about two instances with different parameters, would they be equal or not?

Implement each scenario case:

```java
    @Test
    public void test_correction_test() {
        String name = "pouya";
        String type = "male";
        Play play = new Play(name, type);
        assertTrue((play.name == name) && (play.type == type));
    }
```

```java
    @Test
    public void test_equailty() {
        Play p1 = new Play("name1", "type1");
        Play p2 = new Play("name2", "type2");
        assertFalse(p1.name == p2.name);
        assertFalse(p1.type == p2.type);
    }
```

```
    @Test
    public void test_equailty2() {
        Play p1 = new Play("name", "type");
        Play p2 = new Play("name", "type");
        assertTrue(p1.name == p2.name);
        assertTrue(p1.type == p2.type);
    }
```

Attention:We could create another test case to check whether assigning different types to the constructor parameters results in an exception. However, to keep the number of tests manageable, we will refuse to doing so.
fter writing some tests for the ”Play” class, our next step will be to implement tests for the ”Performance” class. The code implemented below shows the ”Performance” class:

```
    package dramaplays;

    public class Performance {

        public String playID;
        public int audience;

        public Performance(String playID, int audience) {
            this.playID = playID;
            this.audience = audience;
        }
    }
```

This code will simply get a name as a string called ”PlayID” and ”audience” as number of audiences.
Test cases scenarios:

1. When we create an instance of the ”Performance” object, are the parameters set correctly?
2. If two instances of the ”Play” class are created with the same parameters, will their attributes be equal?
3. What about two instances with different parameters, would they be equal or not?
4. If one argument from each class is equal but other parameters are not. It considers as equal class?

Implementing tests:

```
    @Test
    public void test_assign_true() {
        Performance perf = new Performance("play_id", 1000);
        assertEquals(perf.playID, "play_id");
        assertEquals(perf.audience, 1000);
    }
```

```java
    @Test
    public void test_equailty_class() {
        Performance perf1 = new Performance("play_id1", 1000);
        Performance perf2 = new Performance("play_id2", 2000);
        assertFalse(perf1.playID == perf2.playID);
        assertFalse(perf1.audience == perf2.audience);
    }
```

```java
    @Test
    public void test_equailty_class2() {
        Performance perf1 = new Performance("play_id", 1000);
        Performance perf2 = new Performance("play_id", 1000);
        assertTrue(perf1.playID == perf2.playID);
        assertTrue(perf1.audience == perf2.audience);
    }
```

```java
    @Test void test_equailty_class3(){
        Performance perf1 = new Performance("play_id", 1000);
        Performance perf2 = new Performance("play_id", 1001);
        assertTrue(perf1.playID == perf2.playID);
        assertFalse(perf1.audience == perf2.audience);
    }
```

After creating test cases for this object, we must create some tests for ”Invoice” object where you can see the code in following structure:

```java
    package dramaplays;

    import java.util.List;

    public class Invoice {

        public String customer;
        public List<Performance> performances;

        public Invoice(String customer, List<Performance> performances) {
            this.customer = customer;
            this.performances = performances;
        }
    }
```

This code accepts a string parameter ”customer” representing the customer’s name and a second parameter, which is a list of ”Performance” objects.
We already have some tests for ”Performance” class, therefore since this class has no ”hashCode” available for us it would restrict us having fewer test than before.(From test written for ”Performance” class, we can assume this class are correctly implemented!)
Test cases scenarios:

1. Initializing parameters will be correctly assigned.
2. Two equal instances with same parameters will be equal.
3. Two instances with different parameters are not equal.

Implementing test cases:

```java
    @Test
    public void test_invoice_get_instance() {
        List<Performance> performances = new ArrayList<>();
        Invoice inv  = new Invoice("cust", performances);
        assertTrue(inv.customer == "cust" && inv.performances == performances);
    }
```

```java
    @Test
    public void test_invoice_equlity() {
        String cust1 = "Pouya";
        List<Performance> performances1 = new ArrayList<>();
        performances1.add(new Performance("1", 10));
        Invoice inv1 = new Invoice(cust1, performances1);
        String cust2 = "Shariar";
        List<Performance> performances2 = new ArrayList<>();
        performances2.add(new Performance("2", 11));
        Invoice inv2 = new Invoice(cust2, performances2);
        assertFalse(inv1.customer == inv2.customer && inv1.performances == inv2.
           ↪ performances);
    }
```

```java
    @Test
    public void test_invoice_equlity2() {
        String cust1 = "Pouya";
        List<Performance> performances1 = new ArrayList<>();
        performances1.add(new Performance("1", 10));
        Invoice inv1 = new Invoice(cust1, performances1);
        String cust2 = "Pouya";
        List<Performance> performances2 = new ArrayList<>();
        performances2.add(new Performance("1", 10));
        Invoice inv2 = new Invoice(cust2, performances2);
        assertTrue(inv1.customer == inv2.customer);
    }
```

Notice: Note that we assumed ”Performance” class are correctly impelemented as we have written some test to validate it. Therefore we need to check ”customer” parameter in last code!

1.2 Factor Printer Class

Now we can use three classes were validated in previous section, in ”FactorPrinter” class. Code below has been implemented as follows:

```
    package dramaplays;

    import java.text.NumberFormat;
    import java.util.Locale;
    import java.util.Map;



    public class FactorPrinter {

        public String print(dramaplays.Invoice invoice, Map<String, dramaplays.
           ↪ Play> plays) {
            var totalAmount = 0;
            var volumeCredits = 0;
            var result = String.format("Factor for %s\n", invoice.customer);

            NumberFormat frmt = NumberFormat.getCurrencyInstance(Locale.US);

            for (var perf : invoice.performances) {
                var play = plays.get(perf.playID);
                var thisAmount = 0;

                switch (play.type) {
                    case "tragedy":
                        thisAmount = 40000;
                        if (perf.audience > 30) {
                            thisAmount += 1000 * (perf.audience - 30);
                        }
                        break;
                    case "comedy":
                        thisAmount = 30000;
                        if (perf.audience > 20) {
                            thisAmount += 10000 + 500 * (perf.audience - 20);
                        }
                        thisAmount += 300 * perf.audience;
                        break;
                    default:
                        throw new Error("unknown type: ${play.type}");
                }

                // add volume credits
                volumeCredits += Math.max(perf.audience - 30, 0);
                // add extra credit for every ten comedy attendees
```

```java
                if ("comedy".equals(play.type)) volumeCredits += Math.floor(perf.
                    ↪ audience / 5);

                // print line for this order
                result += String.format("  %s: %s (%s seats)\n", play.name, frmt.
                    ↪ format(thisAmount / 100), perf.audience);
                totalAmount += thisAmount;
            }
            result += String.format("Amount owed is %s\n", frmt.format(totalAmount
                ↪  / 100));
            result += String.format("You earned %s credits\n", volumeCredits);
            return result;
        }

    }
```

First, we will explain what this code does. Then, we will implement some tests to build confidence in its functionality. The ”printer” method will get an ”Invoice” and a Map as second parameter. Map is simply is string representing a name and a ”play” object. The method calculates the total amount owed and volume credits based on the type of play and audience size. It formats these values into a string that includes each performance’s details, the total amount, and the earned credits. The method throws an error if an unknown play type is encountered. If you did not underestand what this method does, don’t worry we will explain in creating test cases:

1. We will test the ”Printer” method by creating an invoice for a play called ”Hamlet”. In this test case, we will generate an invoice for a customer named ”John Doe”. The play is a single tragedy performance of ”Hamlet” with 40 seats (audience members). Test case implemented as follows:

```java
        @Test
        public void tragedy_testing() {
            Performance performance = new Performance("hamlet", 40);

            List<Performance> performances = new ArrayList<> ();
            performances.add(performance);
            Invoice invoice = new Invoice("John Doe", performances);

            Play play = new Play("Hamlet", "tragedy");

            Map<String, Play> plays = new HashMap<>();
            plays.put("hamlet", play);

            FactorPrinter printer = new FactorPrinter();
            String result = printer.print(invoice, plays);

            String expected = "Factor for John Doe\n"
                            + "  Hamlet: $500.00 (40 seats)\n"
                            + "Amount owed is $500.00\n"
                            + "You earned 10 credits\n";

            assertEquals(expected, result);
        }
```

The expected result is: Factor for ”customer”
Hamlet: $(This amount for play) (number of seats)
Amount owed is $(Total amount of plays)
You earned (Volume Credit) Credits
So expected results are:

$$\begin{aligned}
\text{Customer} &= \text{John Doe} \\
\text{Seats} &= 40 \\
\text{Hamlet play amount} &= \$(40000 + 1000 * 10)/100 = \$500.00 \\
\text{Total Amount} &= \$500.00 \\
\text{Credits} &= \max(10, 0) = 10 \qquad (1)
\end{aligned}$$

2. We have seen a test created for a single ”tragedy” play. Now, we will proceed to create a test for a single ”comedy” play. In this scenario, our customer is ”John Smith,” and the play is ”As You Like It,” which is a comedy. This performance has 25 seats. More details about the test are as follows:

```
        @Test
        public void comedy_testing() {
            Performance performance = new Performance("as-like", 25);
            List<Performance> performances = new ArrayList<> ();

            performances.add(performance);

            Invoice invoice = new Invoice("Jane Smith", performances);

            Play play = new Play("As You Like It", "comedy");

            Map<String, Play> plays = new HashMap<>();
            plays.put("as-like", play);

            FactorPrinter printer = new FactorPrinter();
            String result = printer.print(invoice, plays);

            String expected = "Factor for Jane Smith\n"
                            + "  As You Like It: $500.00 (25 seats)\n"
                            + "Amount owed is $500.00\n"
                            + "You earned 5 credits\n";

            assertEquals(expected, result);
        }
```

The exepected results are:

$$\begin{aligned}
\text{Customer} &= \text{John Smith} \\
\text{Seats} &= 25 \\
\text{As You Like It amount} &= \$(30000 + 500 * 5 + 300 * 25 + 10000)/100 = \$500.00 \\
\text{Total Amount} &= \$500.00 \\
\text{Credits} &= \max(-5, 0) + 25/5 = 5
\end{aligned} \tag{2}$$

3. Now, let's combine two plays, each with a different type. "Othello" is a tragedy, while "Twelfth Night" is a comedy. The customer is named "Alice Johnson." The "Othello" performance has 40 seats, and "Twelfth Night" has 35 seats. More details for this test are as follows:

```java
    @Test
    public void test_mulitple_performances() {
        List<Performance> performances = new ArrayList<> ();
        Performance perf1 = new Performance("Othello", 40);
        Performance perf2 = new Performance("Twelfth Night", 35);

        performances.add(perf1);
        performances.add(perf2);

        Invoice invoice = new Invoice("Alice Johnson", performances);

        Play play1 = new Play("Othello", "tragedy");
        Play play2 = new Play("Twelfth Night", "comedy");

        Map<String, Play> plays = new HashMap<>();
        plays.put("Othello", play1);
        plays.put("Twelfth Night", play2);

        FactorPrinter printer = new FactorPrinter();
        String result = printer.print(invoice, plays);

        String expected = "Factor for Alice Johnson\n"
                        + "  Othello: $500.00 (40 seats)\n"
                        + "  Twelfth Night: $580.00 (35 seats)\n"
                        + "Amount owed is $1,080.00\n"
                        + "You earned 22 credits\n";

        assertEquals(expected, result);
    }
```

Expected results are:

$$\begin{aligned}
\text{Customer} &= \text{Alice Johnson}\\
\text{Othello Seats} &= 40\\
\text{Twelfth Night Seats} &= 35\\
\text{Othello amount} &= \$(40000 + 10 * 1000)/100 = \$500.00\\
\text{Twelfth Night amount} &= \$(30000 + 500 * 15 + 300 * 35 + 10000)/100 = \$580.00\\
\text{Total Amount} &= \$1,080.00\\
\text{Credits} &= \max(10, 0) + \max(5, 0) + 35/5 = 22
\end{aligned} \tag{3}$$

4. Now from setting a play to ”unkonwn” play type, we will ensure that Error will occurs.

```java
        @Test
        public void test_unkonwn_play() {
            Performance performance = new Performance("unknown-play", 20);
            List<Performance> performances = new ArrayList<> ();
            performances.add(performance);

            Invoice invoice = new Invoice("Bob Brown", performances);

            Play play = new Play("Unknown Play", "unknown");

            Map<String, Play> plays = new HashMap<>();
            plays.put("unknown-play", play);

            FactorPrinter printer = new FactorPrinter();
            Error exception = assertThrows(Error.class, () -> {
                printer.print(invoice, plays);
            });

            assertEquals("unknown type: ${play.type}", exception.getMessage()
                ↪ );
        }
```

We expect that we got an error massage: unknown type: $play.type.

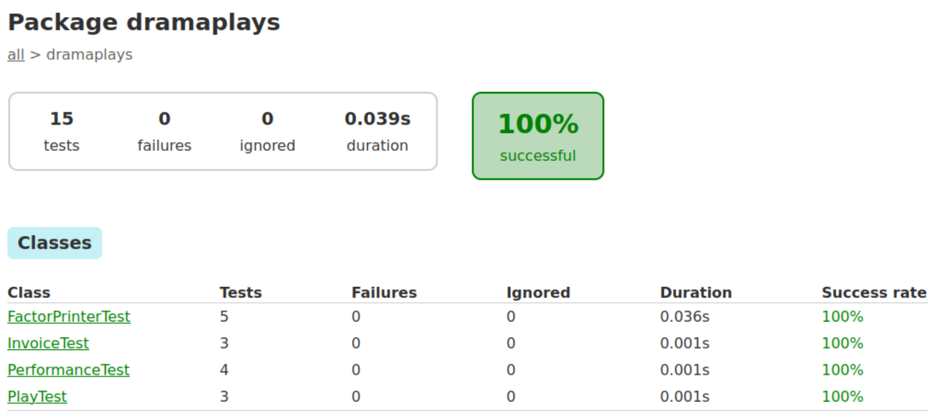
5. The last test would be considered to validate ”Credits” that a customer will achived. So we will consider a comedy play with 55 seats called ”Comedy Play”.Our customer is named ”Clara White”.

```java
        @Test
        public void test_credits() {
            Performance perf = new Performance("comedy-play", 55);
            List<Performance> performances = new ArrayList<> ();
            performances.add(perf);
            Invoice invoice = new Invoice("Clara White", performances);

            Play play = new Play("Comedy Play", "comedy");
        Map<String, Play> plays = new HashMap<>();
        plays.put("comedy-play", play);

        FactorPrinter printer = new FactorPrinter();
        String result = printer.print(invoice, plays);

        String expected = "Factor for Clara White\n"
                        + "  Comedy Play: $740.00 (55 seats)\n"
                        + "Amount owed is $740.00\n"
                        + "You earned 36 credits\n";

        assertEquals(expected, result);
    }
```

The expeted resutls should allign with following characteristics:

$$\begin{aligned}\text{Customer} &= \text{Clara White}\\ \text{Comedy Play seats} &= 55\\ \text{Comedy Play amount} &= \$(30000 + 35 * 500 + 55 * 300 + 10000)/100 = \$740.00\\ \text{Total Amount} &= \$(740.00)\\ \text{Credits} = \max(25, 0) + 55/5 = 36 \end{aligned} \tag{4}$$

The all Characteristics are shown in the results as html where can you see in figure 1:



Package dramaplays

all > dramaplays

15 tests | 0 failures | 0 ignored | 0.039s duration | 100% successful

Classes

Class	Tests	Failures	Ignored	Duration	Success rate
FactorPrinterTest	5	0	0	0.036s	100%
InvoiceTest	3	0	0	0.001s	100%
PerformanceTest	4	0	0	0.001s	100%
PlayTest	3	0	0	0.001s	100%

Figure 1: Results of all testing for 4 classes

Attention:First test is to see if our code works well with a tragedy play. Second test is designed to test comdey play. Third test is designed to validate Total Amount value. Fourth test is designed to see what will happend if type of play is unknown.Last test is designed to ensure the credits are correctly functional.

2 Question 2

First, we will create a Gradle project and add the code to the 'src/main/java/CourseSelection' directory. Next, we will build the project using './gradlew build' to compile the code. After that, we will implement some test cases, with each one being explained in detail.
Provided code is:

```java
package CourseSelection;

import java.util.List;

class Course {
    int id;
    List<Integer> pre;
}

class Record {
    int termId;
    int courseId;
    double grade;
    boolean isMehman;
}

public class CourseSelection{
    public static boolean hasPassedPre(List<Record> rec, Course course) {
        for (int i = 0; i < course.pre.size(); i++) {
            boolean prePassed = false;
            for (int j = 0; j < rec.size(); j++) {
                if (rec.get(j).courseId != course.pre.get(i))
                    continue;
                if (rec.get(j).grade >= 10 && (!rec.get(j).isMehman || rec.get
                ↪ (j).grade >= 12)) {
                    prePassed = true;
                    break;
                }
            } if (!prePassed)
                return false;
        }
        return true;
    }
}
```

Explain code:

Course Object: This class will be used to store the course ID and the IDs of courses that are prerequisites for the current course.

Record: This class is used to record the term ID, course ID, and the grade of the course for the corresponding term. It also includes a boolean field, isMehman, to indicate whether the student is 'Mehman' or not.

CourseSelection: This class includes a method named hasPassedPre. This method takes a list of Record objects and a Course object to identify the course and its prerequisites. By iterating through the prerequisites of the specified course, the method checks if all prerequisites have been passed under the following conditions:

- If the student is not 'Mehman', they must pass all prerequisites with a grade equal to or higher than 10.
- Otherwise they must pass the prerequisites with a grade equal or higher than 12.

Test cases scenarios:

1. The code must return True if 'Record' and prerequisites list in 'Course' obeject is empty.
   Following test has been implemented for this scenario:

```
        @Test
        public void test_has_pas_pre_emty_rec() {
            Course course = new Course();
            course.pre = new ArrayList<> ();
            List<Record> records = new ArrayList<>();
            assertTrue(CourseSelection.hasPassedPre(records, course));
        }
```

2. The code must return True if 'Course''s prerequisites are empty.This secnario test is as follows:

```
        @Test
        public void test_has_pas_pre_emty_course() {
            Course course = new Course();
            course.pre = new ArrayList<> ();
            List<Record> records = new ArrayList<> ();
            Record record = new Record();
            record.termId = 1;
            record.courseId = 101;
            record.grade = 11;
            record.isMehman = false;
            records.add(record);
            assertTrue(CourseSelection.hasPassedPre(records, course));
        }
```

3. The code must return False if one of prerequisites grades are lower than 10 for either is 'Mehman' or not:

```
        @Test
        public void test_has_pas_pre_low_grade() {
            Course course = new Course();
            course.id = 243;
            course.pre = new ArrayList<>();
            course.pre.add(123);
            List<Record> records = new ArrayList<Record>();
            Record record = new Record();
            record.termId = 1;
            record.courseId = 123;
            record.grade = 8;
```

```java
            record.isMehman = false;
            records.add(record);
            assertFalse(CourseSelection.hasPassedPre(records, course));
        }
```

In this scenario, the current course ID is 243, and it has a prerequisite course with ID 123. Therefore, if the student's grade in the prerequisite course is lower than 10, they are not permitted to enroll in course 243.

4. What if the student has higher grade than 10 ?

```java
        @Test
        public void test_has_pas_pre_high_grade() {
            Course course = new Course();
            course.id = 243;
            course.pre = new ArrayList<>();
            course.pre.add(123);
            List<Record> records = new ArrayList<Record>();
            Record record = new Record();
            record.termId = 1;
            record.courseId = 123;
            record.grade = 18;
            record.isMehman = false;
            records.add(record);
            assertTrue(CourseSelection.hasPassedPre(records, course));
        }
```

This is same scenario as before with the difference that grade is 18.Student can enroll 243 course.

5. What happens if a student has exactly a grade of 10?

```java
        @Test
        public void test_has_pas_pre_mid_grade() {
            Course course = new Course();
            course.pre = new ArrayList<>();
            course.id = 243;
            course.pre.add(123);
            List<Record> records = new ArrayList<Record>();
            Record record = new Record();
            record.termId = 1;
            record.courseId = 123;
            record.grade = 10;
            record.isMehman = false;
            records.add(record);
            assertTrue(CourseSelection.hasPassedPre(records, course));
        }
```

Same scenario as before with the difference that student pass 123 course by a grade of 10. The student could enroll to 243 class.

6. In following scenario we assume that some student is ’Mehman’ with a grade lower than 12.

```
        @Test
        public void test_has_pas_pre_isMehman_low_grade() {
            Course course = new Course();
            course.pre = new ArrayList<>();
            course.pre.add(123);
            List<Record> records = new ArrayList<Record>();
            Record record = new Record();
            record.termId = 1;
            record.courseId = 123;
            record.grade = 11;
            record.isMehman = true;
            records.add(record);
            assertFalse(CourseSelection.hasPassedPre(records, course));
        }
```

We expect this student to fail to enroll the 243 Course due their low grade.

7. Next scenario will be same as last test with the differnece that ’Mehman’ student has passed the grade with exactly 12 grade.

```
        @Test
        public void test_has_pas_pre_isMehman_high_grade() {
            Course course = new Course();
            course.pre = new ArrayList<>();
            course.id = 243;
            course.pre.add(123);
            List<Record> records = new ArrayList<Record>();
            Record record = new Record();
            record.termId = 1;
            record.courseId = 123;
            record.grade = 12;
            record.isMehman = true;
            records.add(record);
            assertTrue(CourseSelection.hasPassedPre(records, course));
        }
```

8. What if course’s prerequisites are not an empty list while the ’Record”s list is?

```
        @Test
        public void test_has_pas_pre_empty_rec_has_course() {
            Course course = new Course();
            course.pre = new ArrayList<> ();
            course.id = 243;
            course.pre.add(123);
            List<Record> records = new ArrayList<> ();
            assertFalse(CourseSelection.hasPassedPre(records, course));
        }
```

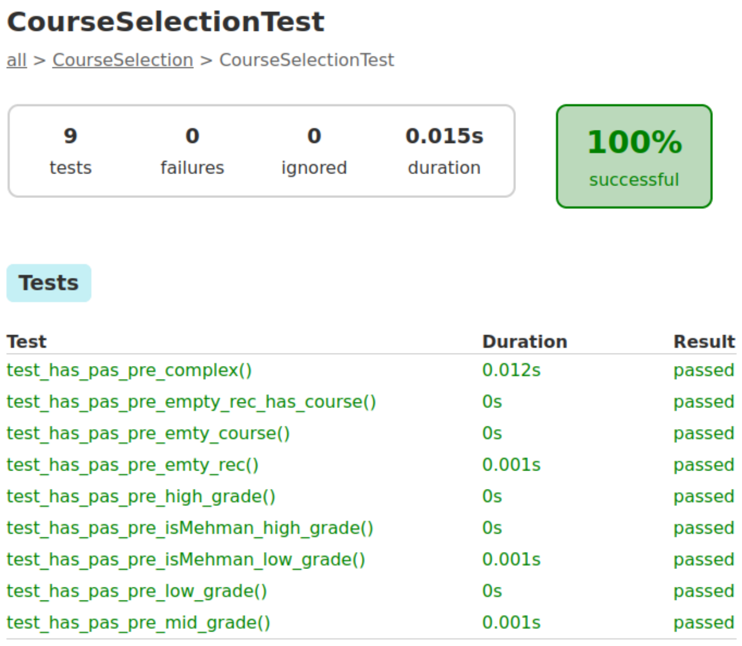
This student did not pass any prerequisites therefore he can not enroll to the 243 course.

9. More complex test with multiple prerequisites:

```
        public void test_has_pas_pre_complex() {
            Course course = new Course();
            course.id = 243;
            course.pre = new ArrayList<> ();
            for(int i = 0; i < 3; i += 1) {
                course.pre.add(i);
            }
            List<Record> records = new ArrayList<> ();
            for(int i = 0; i < 3; i += 1){
                Record record = new Record();
                record.termId = 1;
                record.courseId = i;
                record.isMehman = false;
                record.grade = i + 8;
                records.add(record);
            }
            assertFalse(CourseSelection.hasPassedPre(records, course));
        }
```

This student is forbidden to enroll to the 243 class due he/she has 2 grades lower than 10!

The result of these tests are shown in figure 2:



CourseSelectionTest

all > CourseSelection > CourseSelectionTest

9	0	0	0.015s	100%
tests	failures	ignored	duration	successful

Tests

Test	Duration	Result
test_has_pas_pre_complex()	0.012s	passed
test_has_pas_pre_empty_rec_has_course()	0s	passed
test_has_pas_pre_emty_course()	0s	passed
test_has_pas_pre_emty_rec()	0.001s	passed
test_has_pas_pre_high_grade()	0s	passed
test_has_pas_pre_isMehman_high_grade()	0s	passed
test_has_pas_pre_isMehman_low_grade()	0.001s	passed
test_has_pas_pre_low_grade()	0s	passed
test_has_pas_pre_mid_grade()	0.001s	passed

Figure 2: Test Results for 9 different scenarios

Note that the tests above have thoroughly covered each branch and behavioral aspect of the code.

3 Question 3

Test 1:

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

This test has lack of assertion.The test simply print out the expected and actual results instead of asserting whether they match.In unit testing, we sould use assertions to automatically verify that the code behaves as expected. However by using 'System.out.println', the test requires manual insepction of the output, which is not a efficient way to verify the correctness of the code.
Probable solution:

```
    @Test
    public void testA() {
        Integer result = new SomeClass().aMethod();
        assertEquals(Integer.value(10), reuslt);
    }
```

Test 2:

```
    @Test
    public void testC() expects Exception {
        int badInput = 0;
        new AnotherClass().process(badInput);
    }
```

There is no 'expects' keyword in JUnit.Instead, we can use 'assertThrows' method to check for exceptions.One probable solution for this is as follows:

```
    @Test
    public void testC() {
        int badInput = 0;
        assertThrows(Exception.class, () -> {
            new AnotherClass().process(badInput);
        })
    }
```

Using 'assertThrows' in the test will asserts that the code inside the lambda expression throws and exception of the specified type.
Test 3:

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

In first test we have again lack of assertions. We can assume that Configuration and ResourceManager has two method called 'isInitialized' and then have following assertions for them:

```
    Configuration.initialize();
    ResourceManager.initialize();
    assertTrue(Configuration.isInitialized());
    assertTrue(ResourceManager.isInitialized());
```

The second test has more issues.First this test assumes that 'ResourceManager' has been initialized before checking if a resource is available.If 'ResourceManager.initialize()' is not called before this test, it might fail.
Second reason is not verifing if 'exampleResource' is expected to be available. This might lead to fail since it depend on initial state.
For solving the issue we can use @BeforeEach to before all tests do something for us:

```
    @BeforeEach
    public void setUp() {
        Configuration.initialize();
        ResourceManager.initialize();
    }
    @Test
    public void testResourceAvailability() {
        boolean isResourceAvailable = ResourceManager.isResourceAvailable("
           ↪ exampleResource");
        assertTrue(isResourceAvailable, "The exampleResource should be available")
    }
```

4 Question 4

Yes, we can use JUnit to test multi-threaded programs, although it requires careful design to ensure the tests are realibale and meaningful.
Multi-threaded programs can have issues such as race conditions, deadlocks, and concurrency bugs, which are offen non-determinstic and hard to reproduce!
Note that Junit itself does not have built-in support specifcally for multi-threaded testing. But we can use some frame wordks to aid us to test them.
We can use 'ExecutorService' and 'CountDownLatch'.These are som classes from 'java.util.concurrent' package where can help us to manage and synchronize threads in tests. We can also use assertions and timeouts to check the final state of the program after threads have completed their execution. Timouts will be used to detect if a test is hanging due to deadlocks or not.They are other frameworks which can be helpful for us to deal with multi-threads that we are avoid to explain(like JCTools and etc.)
Attention: Yes, JUnit can be effective for testing multi-threaded programs, but it requires careful setup and additional tools to manage the complexity of concurrency. However, testing such programs is generally not recommended. Instead, formal methods are typically used to handle these complexities.