Section 1: True/False

1. You can have private methods in an interface. **[FALSE]**

2. In a class that implements an interface, the methods that are defined in the interface must have the @Override keyword. **[FALSE]**

3. Given an n-digit number where each digit is either a 0 or 1, the value of the number interpreted in base 10 (decimal) will always be greater than or equal to the value of the number interpreted in base 2 (binary). **[TRUE]**

4. Even numbers, when converted to binary, have an LSB of 1. **[FALSE]**

5. You can use bitwise operators like ^ (XOR) on characters. **[TRUE]**

6. This test will compile: **[True or False we will accept both]**

```
@Test
public void test1() {
    double expected = 4.0;
    double actual = 2.0 + 2.0;
    assertEquals(expected, actual);
}
```

7. This test will compile and pass: **[FALSE]**

```
@Test
public void test1() {
    double radius = 5.0;
    Circle circle = new Circle(radius);
    double expected = 2 * Math.PI * circle.radius;
    assertEquals(expected, circle.circumference, 0.05);
}

public class Circle {
    private double radius;
    private double circumference;

    public Circle(double radius) {
        this.radius = radius;
        this.circumference = 2 * Math.PI * this.radius;
    }
}
```

8. Static methods can use the *this* keyword in them. **[FALSE]**

9. These two snippets of code are equivalent: **[FALSE]**

```
// Snippet 1
Car newCar = new Car("Ford", "Q5", 2019);
Car myCar = new Car("Ford", "Q5", 2019);
// Snippet 2
Car newCar = new Car("Ford", "Q5", 2019);
Car myCar = newCar;
```

10. Suppose DiningHall is an interface, and Hill is a class that implements DiningHall. This code will compile: Examine the following snippets below:

```
Hill hall = new DiningHall();
```

```
DiningHall hill = new Hill();
```

Which of the following is true?

Choice 1 of 4: Snippet 1 compiles; Snippet 2 does not compile
Choice 2 of 4: Snippet 2 compiles; Snippet 1 does not compile
Choice 3 of 4: Both snippets compile
Choice 4 of 4: Neither snippet compiles

11. You can have multiple assert functions in one JUnit test. **[TRUE]**

12. A test will fail if it expects a NullPointerException but the code that runs raises an IllegalArgumentException. **[TRUE]**

13. Getter and setter methods should be kept private. **[FALSE]**

14. Static fields are unique to each instance of a variable. **[FALSE]**

Section 2: Long Fill in the Blank: Taking Attendance

A number of students in CIS 110 are involved in a public advocacy group, and will be taking a trip across the country next week. This means they'll miss recitation, but they will be excused.

The names of students who will be excused are stored in a List<String> (java's implementation of a list of strings) called excusedStudents, and a typical recitation roster is stored as a List<String> of names called roster. excusedStudents contains all excused students in the entire class, but the roster is for a particular recitation, so it may be possible that excusedStudents contains a name that's not present in roster.

Help complete a function List<String> excuseFromRoster(List<String> excusedStudents, List<String> roster) that removes all student names contained in excusedStudents from roster.

- For each student you remove from the roster, you should print out "Removed <STUDENT_NAME>".
- If there is a name in excusedStudents that is not in roster, then you should print the message "Ignored <STUDENT_NAME>".

Hint: remember the Java List functions

```
import java.util.*;

public void excuseFromRoster(List<String> excusedStudents, List<String> roster)
{
    //iterate through entire list of excused students
    for (int i = 0; i < excusedStudents.__1__; i++) {
        //retrieve student at index i
        String excusedStudent = __2__;

        //boolean for if we have yet to find the excused student in roster
        boolean notFound = __3__;

        //iterating through roster list
        for (int j = 0; notFound __4__ j < __5__; j++) {
            //determine if student from above is equal to this student in
roster
            if (excusedStudent.equals(__6__)) {
                __7__;
                notFound = __8__;
                System.out.println("Removed " + excusedStudent);
            }
        }
        if (notFound) {
```

```
            System.out.println("Ignored " + excusedStudent);
        }
    }
}
```

As an example, assume that roster is the list [Harry, Eric, James, Lena], and excusedStudents is the list [James, Harry, Maya]. Calling excuseFromRoster(excusedStudents, roster); should modify roster to become [Eric, Lena]. It should also print out the following:

Removed James
Removed Harry
Ignored Maya

Solution

1 --> size()
2 --> excusedStudents.get(i)
3 --> true
4 --> &&
5 --> roster.size()
6 --> roster.get(j)
7 --> roster.remove(j)
8 --> false (or !notFound)

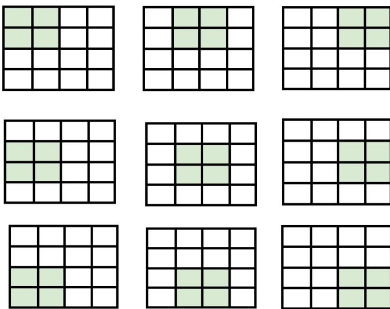
```
import java.util.*;

public void excuseFromRoster(List<String> excusedStudents, List<String> roster)
{
    for (int i = 0; i < excusedStudents.size(); i++) {
        String excusedStudent = excusedStudents.get(i);
        boolean notFound = true;
        for (int j = 0; j < roster.size() && notFound; j++) {
            if (excusedStudent.equals(roster.get(j))) {
                roster.remove(j);
                notFound = false;
                System.out.println("Removed " + excusedStudent);
            }
        }
        if (notFound) {
            System.out.println("Ignored " + excusedStudent);
        }
```

```
    }
}
```

Section 3: Buggy Code

Unbeknownst to many, Becca owns a massive gold field out in France. The field isn't actually a gold field though, it's a truffle field - the most expensive food on Earth. Becca's field is a 4 by 4 grid (2D array) of land, and at each specific pair of indices lies a Truffle object. The Truffle object has two fields, a String type (there are three types of truffles, "white" truffles, "black" truffles, and "burgundy" truffles) and an int price (the amount the truffle can be sold for).

This may sound like good money, but Becca has an issue - she only has enough energy to dig up the truffles in one singular, contiguous 2 by 2 block of land. Becca has a truffle sniffing pig that can thankfully sniff up the truffles and determine what a truffle is worth without having to dig it up. Note that in a 4 by 4 grid, there are 9 contiguous 2 by 2 blocks of land. Below are pictures depicting the 9 possible blocks she can choose to dig up truffles from.



Becca has written a program to help her find the maximum profit she can earn, and even wrote some tests too! Sadly, her code is not working, and she could use your help in finding the bugs.

Here is Becca's **correct** code for a simple Truffle object (there are no bugs in this class):

```
public class Truffle {
    private String type; //can be "black", "white", or "burgundy"
    private int price;

    //constructor, take in a type and price
    public Truffle(String type, int price) {
        if (!type.equals("black") && !type.equals("white")
            && !type.equals("burgundy")) {
            throw new IllegalArgumentException("Illegal truffle type.");
        }

        this.type = type;

        if (price < 0) {
```

```
            throw new IllegalArgumentException("Illegal truffle price.");
        }

        this.price = price;
    }

    //getter method for price
    public int getPrice() {
        return this.price;
    }

    //getter method for type
    public String getType() {
        return this.type;
    }
}
```

Here is Becca's buggy program and tests:

```
1. public class TruffleField {
2.    private Truffle[][] truffleField; //private Truffle-2d array field
3.
4.    //constructor
5.    public TruffleField(Truffle[][] truffleField) {
6.        if (null.equals(truffleField)) {
7.            throw new IllegalArgumentException("Null input");
8.        }
9.        //initialize the truffle field
10.       this.truffleField = truffleField;
11.    }
12.
13.   //getter method for truffleField
14.   public Truffle[][] getField() {
15.       return this.truffleField;
16.   }
17.
18.    //returns the max Profit Becca can earn from single 2x2 block
19.    public int maxProfit(){
20.        int maxProfit = 0;
21.        //iterate through the rows and then the columns
22.        for (int row = 0; row < truffleField.length; row++) {
23.            for (int col = 0; col < truffleField[row].length; col++) {
```

```
24.                 //handles out of bounds errors for each 2x2 block
25.                 if (row + 1 < truffleField.length ||
26.                      col + 1 < truffleField[row].length) {
27.                     int twoByTwoProfit = 0;
28.
29.                     //iterate through the 2 by 2 block here
30.                     for (int i = 0; i < 2; i++) {
31.                         for (int j = 0; j < 2; j++) {
32.                            //add truffle's price to profit variable
33.                            int prof = truffleField[row+i][col+j].price;
34.                            twoByTwoProfit += prof;
35.                         }
36.                     }
37.
38.                     if (twoByTwoProfit > maxProfit) {
39.                         maxProfit = twoByTwoProfit;
40.                     }
41.                 }
42.             }
43.         }
44.
45.         return maxProfit;
46.     }
47. }
```

```
48. import static org.junit.Assert.*;
49. import org.junit.*;
50.
51. public class TruffleFieldTests {
52.    //test constructor for bad (null) input
53.    @Test
54.    public void testBrokenConstructor() {
55.        TruffleField broken = new TruffleField(null);
56.    }
57.
58.    //test for a truffleField with every truffle having a price of 2K
59.    @Test
60.    public void testMaxProfit8000() {
61.        //create the actual truffleField
62.        Truffle[][] test = new Truffle[4][4];
63.        for (int i = 0; i < 4; i++) {
64.            for (int j = 0; j < 4; j++) {
```

```
65.                 test[i][j] = new Truffle("burgundy", 2000);
66.             }
67.         }
68.
69.         //make sure the profit is 8000
70.         TruffleField testField = new TruffleField(test);
71.         assertTrue(8000, testField.maxProfit());
72.
73.         //test for side effects (underlying 2d-arrays are equivalent)
74.         assertEquals(test, testField.getField());
75.     }
76. }
```

There are 6 errors in total: 3 in the TruffleFieldTests class and 3 in TruffleField class. Some are compile time errors and some are logic errors (i.e. code that incorrectly solves a specific task / is incorrectly implemented). Please include all errors in the following format, with each error on its own line:

Line Number / Description of error / Correct Line of Code
(Example: Line 42 / Doesn't parse string x for integer / Integer.parseInt(x))

Solution

- Line 6 / Null pointer error / field == null
- Line 25 / Faulty Logic / Change || to &&
- Line 33 / Access a private field without the getter / Change .price to .getPrice()
- Line 53 / Missing the (expected = IllegalArgumentException.class) / add it back!
- Line 71 / Should be assert Equals rather than assert true
- Line 74 / Should be assert Array Equals rather than assert equals

Section 5: Long Coding - CIS110IFY

TAs from CIS want to start a competitor to Spotify, called CIS110IFY. So far, they were able to write the classes for a CIS110IFY (like a Playlist) and a Song. A Song is like a Node: it stores a String for the title and a reference to a next Song. A CIS110IFY object is implemented as a linked sequence of Song objects, starting from the instance variable firstSong.

The TAs wrote the Song class as well as the queueSong(), disableLoop (but not its helper) and start() methods for CIS110IFY. Unfortunately, they got sidetracked making tier lists of Marvel movies, and they need you to finish the enableLoop() and disableLoopHelper() methods for the CIS110IFY class. We have provided a test class CIS110IFYTest.java that includes some example tests and will include the tests you write later. You are also welcome to study and reuse code from the rest of CIS110IFY.java (for example, the start() method is for your reference only and is never actually used in the methods you write).

```
public class Song {
    public String title;
    public Song next;

    public Song(String title) {
        this.title = title;
    }

    public Song(String title, Song next) {
        this.title = title;
        this.next = next;
    }

    public void play() {
        System.out.println("Now playing: " + title);
    }
}
```

```
public class CIS110IFY {
    public Song firstSong;

    /*
```

```java
     * Adds a song "nextSong" to the end of the linked list.
     */
    public void queueSong(Song nextSong) {
        // adding a song to an empty linked list
        if (firstSong == null) {
            firstSong = nextSong;
            return;
        }

        // adding a song to a non-empty linked list
        Song currentSong = firstSong;
        while (currentSong.next != null) {
            currentSong = currentSong.next;
        }
        currentSong.next = nextSong;
    }

   /*
     * Starting from firstSong, play each song in the
linked list one after the other.
     * Even if the Linked List is longer, or is set to loop, then the linked
list should
     * only play up to 10 songs before restarting.
     * (Buy CIS110IFY Premium to enable longer playlists!)
     */
    public void start() {
        Song currentSong = firstSong;
        int songCount = 0;
        while (currentSong != null && songCount < 10) {
            currentSong.play();
            songCount++;
            currentSong = currentSong.next;
        }
    }

    /*
     * Changes the Linked List so that the final Song points
     * to the first Song in the Linked List.
     */
    public void enableLoop() {
        // TODO
    }
```

```
   /*
    * Changes the Linked List so that the last Song that plays before the
Linked List
    * repeats instead points to null.
    * In other words, stop the linked list from looping.
    */
    public void disableLoop() {
        disableLoopHelper(firstSong);
    }

    public void disableLoopHelper(Song currentSong) {
        // TODO
    }


}
```

```
import static org.junit.Assert.*;
import org.junit.*;

public class CIS110IFYTest{
    @Test
    public void testQueueSongEmpty() {
        Song paprika = new Song("Paprika");
        CIS110IFY p = new CIS110IFY ();
        p.queueSong(paprika);

        assertEquals(paprika, p.firstSong);
    }

    @Test
    public void testQueueSongTwice() {
        Song paprika = new Song("Paprika");
        Song certainty = new Song("Certainty");
        CIS110IFY p = new CIS110IFY ();
        p.queueSong(paprika);
        p.queueSong(certainty);

        assertEquals(paprika, p.firstSong);
        assertEquals(certainty, p.firstSong.next);
```

```
    }
}
```

testEnableLoop()

Complete the test case for testEnableLoop() by following these steps:

1. Your test should use queueSong() on the three provided songs so that your CIS110IFY has the following structure:
2. Call enableLoop() on your CIS110IFY object. The sequence of songs should now match the following image.
3. Verify that your CIS110IFY has the correct structure by writing an assertion statement. Your assertion should be checking that the last song in the linked list points to the first song. You don't have to worry about testing for other side-effects.

Solution

```
@Test
    public void testEnableLoop() {
        Song amoeba = new Song("Amoeba");
        Song nobody = new Song("Nobody");
        Song aja = new Song("Aja");

        CIS110IFY p = new CIS110IFY();

        //SOLUTION FOR TestEnableLoop
        p.queueSong(amoeba);
        p.queueSong(nobody);
        p.queueSong(aja);
        // should traverse through p and set the last song, aja,
        // to point its next to firstSong, which is amoeba.
        p.enableLoop();
        assertEquals(aja.next, amoeba);
    }
```

enableLoop() Implementation

Write enableLoop().

Before calling enableLoop() on a CIS110IFY, the final Song in the linked sequence of songs has its next reference be set to null. enableLoop() should set the next instance variable of the final Song to point to the firstSong of the CIS110IFY. In this way, calling start() on the looped CIS110IFY will repeat the songs until 10 songs have been played.

```
public void enableLoop() {
    //code to implement
}
```

Write your enableLoop() function here:

Solution

```
public void enableLoop() {
       if (firstSong == null) {
           return;
       }
       Song currentSong = firstSong;
       while (currentSong.next != null) {
           currentSong = currentSong.next;
       }
       currentSong.next = firstSong;
}
```

testDisableLoop()

Complete the test case for testDisableLoop() by following these steps

1. Your test should use queueSong() on the three provided songs so that your CIS110IFY has the following structure:
2. Call disableLoop() on your CIS110IFY object. The sequence of songs should now match the following image:
3. Verify that your CIS110IFY has the correct structure by writing an assertion statement. Your assertion should be checking that the last song in the linked list has a null pointer, and also that the firstSong instance variable still points to the correct song. You don't have to worry about testing for other side-effects.

Solution

```
@Test
    public void testDisableLoop() {
        Song amoeba = new Song("Amoeba");
        Song nobody = new Song("Nobody");
        Song aja = new Song("Aja", amoeba); // the last song points to the
firstSong

        CIS110IFY p = new CIS110IFY();
        // SOLUTION FOR TestDisableLoop
        p.queueSong(amoeba);
        p.queueSong(nobody);
        p.queueSong(aja);

        // should traverse through p and set the last song, "Aja",
        // to point its next to null.
        p.disableLoop();
        assertNull(aja.next);
        assertEquals(amoeba, p.firstSong);
    }
```

disableLoopHelper() Implementation

Before calling disableLoop() on a CIS110IFY, there will be one Song in the sequence of songs that has its next reference set to firstSong. disableLoop() should find this last Song that points to firstSong and set its next instance variable to be null. The catch? **You must implement this function recursively. Solutions that are not recursive/use any kind of loop will receive ZERO points.**

To accomplish this, you will be completing the disableLoopHelper() method, which is called by disableLoop() on the initial input firstSong.

Again, you must implement this function recursively. Iterative solutions will be awarded ZERO points.

```
public void disableLoop() {
    if (firstSong == null) {
        return;
    }
```

```
    disableLoopHelper(firstSong);
}


public void disableLoopHelper(Song currentSong) {
    //code to implement
}
```

Write your disableLoopHelper() function here:

Solution

```
public void disableLoopHelper(Song currentSong) {
      if (currentSong.next == firstSong) { //remember assume that a loop
exists
            currentSong.next = null;
      } else {
            disableLoopHelper(currentSong.next);
      }
}
```

Section 4: Short Coding - Too Cool 4 When2Meet

With the Spring semester coming soon, the head TAs for CIS 110 are trying to find a time to host staff meetings. Given the CIS gods that they are, they refuse to use When2Meet and want to write their own program to calculate how many TAs are available at each possible time.

To this extent, they have made the following TA Class:

```
public class TA {
    private String name;
    /**
     * 8 rows, 5 columns
     * The rows represent hours starting at 9am, 10am, ... 4pm
     * The columns represent Mon, Tues, Wed, Thurs, Fri
     * If an element is true, the TA is available at that hour on that day
     * availability[2][3] is true if the TA is available 11:00AM-12:00PM on
Thursday
     */
    private boolean[][] availability;

    public TA(String name, boolean[][] availability) {
        this.name = name;
        this.availability = availability;
    }

    public boolean[][] getAvailability() {
        return this.availability;
    }
}
```

They've also written a Staff class which you will help them complete. This is what they have so far!

```
public class Staff {

    private TA[] tas;

    public Staff(TA[] tas) {
        if (tas == null) {
            throw new IllegalArgumentException();
        }
        this.tas = tas;
    }
```

```
    public int[][] countAvailability() {
        // TODO
    }

}
```

Notably, the Staff class has one instance variable: an array of TA objects that contains all the TAs for the class.

They've asked you to write a function which counts how many TAs (from the tas field) are available at each possible hour/day. So, the output of this method is an int[][] with 8 rows and 5 columns, where the element in the ith row and jth column is the number of TAs (in the field) available at ith hour and on the jth day. You may assume that the input array is non null and non empty.

Consider this example with 2 TAs.

```
TA shivin = new TA("Shivin", new boolean[][] {
            {false, false, false, false, true},
            {false, false, false, false, false},
            {false, false, false, false, true},
            {false, false, false, false, false},
            {false, false, false, false, true},
            {false, false, false, false, true},
            {false, false, false, false, true},
            {true, false, false, false, false}
            });

TA gian = new TA("Gian", new boolean[][] {
            {false, false, false, false, true},
            {false, false, false, false, false},
            {false, false, false, false, true},
            {false, false, false, false, false},
            {false, false, false, false, false},
            {false, false, true, false, true},
            {false, false, false, false, true},
            {false, false, false, false, false}
            });
TA[] taArray = {shivin, gian};
```

staff.countAvailability() for the above example would return the following 2D array:

```
{{0, 0, 0, 0, 2},
 {0, 0, 0, 0, 0},
 {0, 0, 0, 0, 2},
 {0, 0, 0, 0, 0},
 {0, 0, 0, 0, 1},
 {0, 0, 1, 0, 2},
 {0, 0, 0, 0, 2},
 {1, 0, 0, 0, 0}}
```

Please fill in the following function:

```
public int[][] countAvailability() {
    // TODO
}
```

Solution

```
public int[][] countAvailability() {
        // 8 hours a day, 5 days a week
        int[][] output = new int[8][5];
        for (int i = 0; i < this.tas.length; i++) {
            boolean[][] taAvailability = this.tas[i].getAvailability();
            for (int h = 0; h < 8; h++) {
                for (int d = 0; d < 5; d++) {
                    if (taAvailability[h][d]) {
                        output[h][d]++;
                    }
                }
            }
        }
        return availability;
    }
```