exam2-solutions.md

Exam 2

All questions, each paired with its answer. It is probably best to navigate the document using the hyperlinked table of contents to avoid too much scrolling.

True/False

- 1a. You cannot overload constructors in Java. **(False)**
- 1b. The following code prints `"Shivin"` : **(False)**

```
public class Student {
    private String name;
    public Student(String name) {
        name = name;
    }

    public static void main(String[] args) {
        Student shivin = new Student(“Shivin”);
        System.out.println(shivin.name);
    }
}
```

- 1c. The following code prints a memory address: **(False)**

```
public class Student {
    private String name;
    public Student(String name) {
        name = name;
    }
    public String toString() {
        return name;
    }
    public static void main(String[] args) {
        Student shivin = new Student(“Shivin”);
        System.out.println(shivin);
    }
}
```

- 1d. The following code compiles: **(True)**

```
public class Student {
    private String name;
    private Student[] friends;

    public Student(String student) {
        name = student;
        friends[0] = this;
    }
}
```

- 1e. The following code correctly removes the first Node in a Linked List: **(False)**

```
public void removeFirstNode() {
    headNode.next = headNode;
}
```

- 1f. The following code snippet adds a node after the first node: **(False)**

```
public void mystery(int value) {
    Node newNode = new Node(value);
    newNode.next = head;
    head = newNode;
}
```

- 1g. This code snippet prints `"j"` : **(False)**

```
LinkedList sandwich = new LinkedList ();
sandwich.append("j");
sandwich.append("pb");
sandwich.insert(1, "j");
sandwich.insert(1, "pb");
sandwich.remove(1);
System.out.print(test.get(2));
```

- 1h. ArrayLists and LinkedLists are dynamically sized. (**True**)
- 1i. The above template for the list Gifts correctly implements the GiftList interface. **(False)**

```
public interface GiftList {
    public void add(Gift a);
    public void gift(Gift b);
}

public class Gifts implements GiftList {
    public void addGift(Gift a) {
        // *code to add a present to the list*
    }
    public void giveGift(Gift b) {
        // *code to gift a present from the list*
    }
}
```

- 1j. You cannot add an object to a list at a position greater than the size of the list. **(True)**
- 1k. The below tests display the same results: **(True)**

  ```
  public void testIsMammal() {
      String animal = “cat”;
      boolean expected = true;
      boolean actual = isMammal(animal);
      assertEquals(expected, actual);
  }

  public void testIsMammal() {
      Boolean actual = isMammal(“cat”);
      assertTrue(actual);
  }
  ```

- 1l. You write a test case for a function called countAttendance. The test case fails. True or false: there must be a bug in countAttendance. **(False)**
- 1m. If an `IllegalArgumentException` is thrown, this test under this header would pass. **(True)**

  ```
  @Test (expected = IllegalArgumentException.class)
  ```

- 1n. This is a file called `Buzz.java`.

  ```
  public class Buzz {
      private int fizz;
      public Buzz() {
          fizz = int (Math.random() * 10);
      }
  }
  ```

  The following lines will cause a compilation issue when written in a separate class `TestBuzz.java`. **(True)**

  ```
  Buzz b = new Buzz();
  System.out.println(b.fizz);
  ```

- 1o. This is a file called `Buzz.java`.

  ```
  public class Buzz {
      private int fizz;
      public Buzz() {
          fizz = int (Math.random() * 10);
      }
  }
  ```

  The following lines will cause a compilation issue when written in a separate class `TestBuzz.java`. **(False)**

  ```
  Buzz b = new Buzz();
  System.out.println(b);
  ```

> N.B. 1n and 1o are presented as they appeared in the exam, including a typo in Buzz.java. The typo does not change the answer of the question, as the lines in question from TestBuzz.java are written as intended both times. In the case of 1n, accessing b.fizz would be a compiler error. For 1o, neither line from TestBuzz.java would cause an error.

- 1p. If `a.equals(b)` is `true`, then `a == b` is `true` always. **(False)**
- 1q. If `a == b` is `true`, then `a.equals(b)` is `true` always. **(True)**
- 1r. If a class `Novel` has a method `public static void printSummary()`, and if `book` is an instance of class `Novel`, then calling `book.printSummary()` will cause an error. **(False)**
- 1s. If `TV` is an interface, the following code will compile: `TV tv = new TV();` **(False)**
- 1t. An interface must be implemented by some class to compile. **(False)**

- 1u. If `Grape` & `Blueberry` implement a `Fruit` interface, we can do the following: **(True)**

  ```
  Fruit g = new Grape();
  Fruit b = new Blueberry();
  Fruit[] fruits = {g, b};
  ```

Short Fill in the Blank

- 2a. A **static** field is associated with the class rather than any object. That means that there is only one instance of that field in memory.
- 2b. If the visibility modifier of a field is **private**, then accessing its value outside of its class definition requires the use of a getter.
- 2c. The **constructor** is a special method of a class that creates a new object from that class. It is easy to identify because it shares a name with the class itself.
- 2d. Math.random() and PennDraw.advance() are **static** functions, which explains why each function can be invoked without creating an instance of their respective classes.

> Other reasonable answers were accepted for 2d.

Fill in the Blank Functions

3a:

The TAs are throwing a holiday party and are stringing up lights to celebrate. As they removed each bulb from the box, they gave it a random brightness value. They then connected the lights randomly to each other, either on the bottom of or to the right of each previous light. They need your help extending the chain of lights by providing an array that lists the lights' brightness value from left to right, first going down as far as possible for each bulb. For example, you should return `[10, 15, 16, 17, 25, 1, 2, 0, 7]` from the diagram below.

Assume that the function is initially called with `linkTheLights(head, 0, arr);` Where `head` is the top-left node and `arr` is an integer array with size of the number of lights. Also assume that only bulbs in the top row (10, 15, 1, and 0) can point to another bulb to its right. For example, bulb 16 would never be able to point to a bulb to its right. Do not worry about the syntax of the `PartyLight` class (i.e. `public static class PartyLight`). Even though you should not worry about this syntax, please do not change it for compilation purposes.

```
public static class PartyLight {
    PartyLight right; //pointer to right PartyLight
    PartyLight down; //pointer to down partyLight
    int brightness; //value of light's brightness


    //constructor with right and down pointers + brightness parameter
    PartyLight(PartyLight r, PartyLight d, int v){
        this.right = r;
        this.down = d;
        this.brightness = v;
    }
    //constructor with just brightness parameter
    PartyLight(int v) {
        this.brightness = v;
    }
}


public static void linkTheLights(PartyLight curr, int index, int[] arr){
    if(___(1)___){
        return;
    }
    arr[index] = ___(2)___;
```

```
    index++;

    PartyLight n = curr;

    while(___(3)___){
        n = ___(4)___;
        arr[index] = ___(5)___;
        index++;
    }

    linkTheLights(___(6a)___, ___(6b)___, ___(6c)___);

}
```

Solutions:

1. `curr == null`
2. `curr.brightness`
3. `n.down != null`
4. `n.down`
5. `n.brightness`
6. a: `curr.right` , b: `index` , c: `arr`

> On Gradescope, these answers appear in the following order to account for a scrambled output from Canvas: 5, 1, 2, 3, 4, 6c, 6a, 6b. We have accounted for this in grading.

3b:

3b: Michael, the president of UPenn Mathletes, is playing a game with his club. In order to play the game, each mathlete lines up in such a way that there is either a person to their right, or a person down below them. For the sake of this game, every person is identified by their favorite number. The mathletes are great at math, but need your help with a coding problem. They would like to write a function that takes in an array and populates it with the mathletes' favorite numbers from left to right, first going down as far as possible for each mathlete. For example, you should return `[10, 15, 16, 17, 25, 1, 2, 0, 7]` from the diagram below.

Assume that the function is initially called with `findFavNumbers(head, 0, arr);` Where `head` is the top-left mathlete and `arr` is an integer array with size of the number of mathletes playing the game. Also assume that only mathletes in the top row (10, 15, 1, and 0) can point to another mathlete to their right. For example, mathlete 16 would never be able to point to a mathlete to their right. Do not worry about the syntax of the `Mathlete` class (i.e. `public static class Mathlete`). Even though you should not worry about this syntax, please do not change it for compilation purposes.

```
public static class Mathlete {
    Mathlete right; //pointer to right Mathlete
    Mathlete down; //pointer to down Mathlete
    int favNumber; //value of Mathlete's favorite number

    //constructor with right and down pointers + value of favNumber parameter
    Mathlete(Mathlete r, Mathlete d, int v){
        this.right = r;
        this.down = d;
        this.favNumber = v;
    }

    //constructor with just value parameter
    Mathlete(int v) {
        this.favNumber = v;
    }
}
```

```
public static void findFavNumbers(Mathlete curr, int index, int[] arr){
    if(___1____){
        return;
    }
    arr[index] = _______2_____;
    index++;

    Mathlete n = curr;

    while(_____3____){
        n = ___4__;
        arr[index] = _____5____;
        index++;
    }

    findFavNumbers(__6a__, __6b__, __6c__);
}
```

Solutions:

1. `curr == null`
2. `curr.favNumber`
3. `n.down != null`
4. `n.down`
5. `n.favNumber`
6. a: `curr.right` , b: `index` , c: `arr`

> On Gradescope, these answers appear in the following order to account for a scrambled output from Canvas: 1, 2, 3, 4, 5, 6a, 6c, 6b. We have accounted for this in grading.

3c:

When not TAing CIS 110, Jules likes to spend her free time searching for buried treasure. In Denver, she came across a huge field with treasure everywhere! However, she noticed that the treasure chests were arranged in such a way that for every chest, there is either a chest to the right, or a chest down below it. On top of every chest is a number etched into the wood that denotes how many jewels Jules will find in the chest. Jules would like to write a function that takes in an array and populates it with the number of jewels in the chests from left to right, first going down as far as possible for each chest. For example, you should return `[10, 15, 16, 17, 25, 1, 2, 0, 7]` from the diagram below.

Assume that the function is initially called with `treasureHunt(head, 0, arr);` Where `head` is the top-left treasure chest and `arr` is an integer array with size of the number of treasure chests on the field. Also assume that only chests in the top row (10, 15, 1, and 0) can point to another chest to its right. For example, chest 16 would never be able to point to a chest to its right. Do not worry about the syntax of the `TreasureChest` class (i.e. `public static class TreasureChest`). Even though you should not worry about this syntax, please do not change it for compilation purposes.

```
public static class TreasureChest {
    TreasureChest right; //pointer to right treasure chest
    TreasureChest down; //pointer to down treasure chest
    int jewels; //value of jewels in treasure chest

    //constructor with right and down pointers + jewel value
    TreasureChest(TreasureChest r, TreasureChest d, int v){
        this.right = r;
        this.down = d;
        this.jewels = v;
    }

    //constructor with just jewel value parameter
    TreasureChest(int v) {
        this.jewels = v;
    }
}


public static void treasureHunt(TreasureChest curr, int index, int[] arr){
    if(____1___){
        return;
    }
    arr[index] = ______2______;
    index++;

    TreasureChest n = curr;

    while(_____3____){
        n = __4___;
        arr[index] = ______5___;
        index++;
    }

    treasureHunt(__6a__, __6b__, __6c__);
}
```

Solutions:

1. `curr == null`
2. `curr.jewels`
3. `n.down != null`
4. `n.down`
5. `n.jewels`
6. a: `curr.right` , b: `index` , c: `arr`

4a:

Rasterization is a technique in computer graphics used to color in polygons. Nick would love to build his own rasterizer that takes in a 2D `int[][]` array representing a polygon, and modify the input array so that the polygon is fully colored in. The input comes in the following format:

```
int[][] polygon = {
    {0, 1, 0, 1, 0},
    {0, 14,  0,  0, 0, 0, 14},
    {0, 0, 1, 0, 0, 1},
    {0, 0, 3, 0}
    {0, 0, 44, 0, 44}
}
```

Each row in our polygon contains either 1 positive integer or 2 positive identical integers representing colors. There is only one color in a row. If there is only one integer in the row, we do not have to change anything. If there are 2 integers however, then we have to fill in the space between the 2 with the same integer value. Thus, the output looks like this:

```
int[][] polygon = {
    {0, 1, 1, 1, 0},
    {0, 14,  14,  14, 14, 14, 14},
    {0, 0, 1,1, 1, 1},
    {0, 0, 3, 0}
    {0, 0, 44, 44, 44}
}
```

Notice that the rows are not of the same length: the input may be a ragged matrix. Fill in the blanks of the following function:

```
public static void rasterizer(int[][] polygon) {
        for (int row = 0; row < polygon.length; row++) {
                int numCounter = 0;
                int color = 0;
                for (int col = 0; __1___; col++) {
                        if (polygon[row][col] > _2__) {
                                numCounter++;
                                color = polygon[row][col];
                        }
                }
                if (____3____) {
                        continue;
                }
                boolean colorIn = false;
                for (int col = 0; col < polygon[row].length; col++) {
                        if (__4____ && !colorIn) {
                                colorIn = true;
                        } else if (polygon[row][col] == 0 && colorIn) {
                                ____5______ = color;
                        } else if _____6_______ {
                                break;
                        }
                }
        }
}
```

Solution:

1. `col < polygon[row].length`
2. `0`
3. `numCounter < 2`
4. `polygon[row][col] > 0` (also, `!= 0` works too).
5. `polygon[row][col]`
6. `polygon[row][col] > 0 && colorIn`

4b:

Michelle is trying to set up her schedule for next semester. She is representing her schedule as a 2D array of integers, with each row representing a day. Each `int` in a row represents an hour: 0 if she has no class then or a positive course number if she has class. However, when she tried to download her schedule, it only included the beginning and end of each class! Help Michelle write a function to fill in the gaps in her schedule with the proper course numbers.

The input comes in the following format:

```
int[][] schedule = {
    {0, 1, 0, 1, 0},
    {0, 14,  0,  0, 0, 0, 14},
    {0, 0, 1, 0, 0, 1},
    {0, 0, 3, 0}
    {0, 0, 44, 0, 44}
}
```

Each row in our schedule contains either 1 positive integer or 2 positive identical integers representing courses. She only has one class in a day. If there is only one integer in the row, we do not have to change anything. If there are 2 integers however, then we have to fill in the space between the 2 with the same integer value. Thus, the output looks like this:

```
int[][] schedule = {
    {0, 1, 1, 1, 0},
    {0, 14,  14,  14, 14, 14, 14},
    {0, 0, 1,1, 1, 1},
    {0, 0, 3, 0}
    {0, 0, 44, 44, 44}
}
```

Notice that the rows are not of the same length: the input may be a ragged matrix. Fill in the blanks of the following function:

```
public static void scheduleFiller(int[][] schedule) {
        for (int row = 0; row < schedule.length; row++) {
                int numCounter = 0;
                int class = 0;
                for (int col = 0; __1___; col++) {
                        if (schedule[row][col] > _2__) {
                                numCounter++;
                                class = schedule[row][col];
                        }
                }
                if (____3____) {
                        continue;
                }
                boolean inClass = false;
                for (int col = 0; col < schedule[row].length; col++) {
                        if (___4___ && !inClass) {
                                inClass = true;
                        } else if (schedule[row][col] == 0 && inClass) {
                                ____5______ = class;
                        } else if _____6_______ {
                                break;
                        }
                }
        }
}
```

Solution:

1. `col < schedule[row].length`
2. `0`
3. `numCounter < 2`
4. `schedule[row][col] > 0` (also, `!= 0` works too).
5. `schedule[row][col]`
6. `schedule[row][col] > 0 && inClass`

4c:

4c: Liam is trying to arrange a piece of music for an a cappella performance. The song he is working on is represented as a 2D array of integers, with each row representing a different vocalist's part. Each `int` in a row represents a beat: a rest (0) if the vocalist does not sing at that beat or a note (positive number) if the vocalist sings at that beat. However, Liam wants the song to be smoother, so he wants to fill in the gaps between notes if they are the same. Help him write a function to accomplish this!

The input comes in the following format:

```
int[][] song = {
    {0, 1, 0, 1, 0},
    {0, 14,  0,  0, 0, 0, 14},
    {0, 0, 1, 0, 0, 1},
    {0, 0, 3, 0}
    {0, 0, 44, 0, 44}
}
```

Each row in our song contains either 1 positive integer or 2 positive identical integers representing notes. There is only one note in a row. If there is only one integer in the row, we do not have to change anything. If there are 2 integers however, then we have to fill in the space between the 2 with the same integer value. Thus, the output looks like this:

```java
int[][] song = {
    {0, 1, 1, 1, 0},
    {0, 14,  14,  14, 14, 14, 14},
    {0, 0, 1, 1, 1, 1},
    {0, 0, 3, 0}
    {0, 0, 44, 44, 44}
}
```

Notice that the rows are not of the same length: the input may be a ragged matrix. Fill in the blanks of the following function:

```java
public static void songSmoother(int[][] song) {
        for (int row = 0; row < song.length; row++) {
                int numCounter = 0;
                int note = 0;
                for (int col = 0; __1___; col++) {
                        if (song[row][col] > _2__) {
                                numCounter++;
                                note = song[row][col];
                        }
                }
                if (____3____) {
                        continue;
                }
                boolean betweenNotes = false;
                for (int col = 0; col < song[row].length; col++) {
                        if (___4___ && !betweenNotes) {
                                betweenNotes = true;
                        } else if (song[row][col] == 0 && betweenNotes) {
                                ___5_______ = note;
                        } else if _____6_______ {
                                break;
                        }
                }
        }
}
```

Solution:

1. `col < song[row].length`
2. `0`
3. `numCounter < 2`
4. `song[row][col] > 0` (also, `!= 0` works too).
5. `song[row][col]`
6. `song[row][col] > 0 && betweenNotes`

Long Coding

5a: Vaccination for the Nation

Pfizer & Moderna have both developed COVID-19 vaccines that are ready for distribution. Due to the lack of coordination, both companies are maintaining separate queues that are both modelled as List.

Each person must receive 2 shots of a COVID-19 vaccine - this can be both Moderna, both Pfizer or 1 Moderna + 1 Pfizer (company doesn't matter as long as they get 2 shots of a vaccine). Both companies administer only 1 shot at a time. At a given point of time, a Person can be in both queues **but cannot appear multiple times in the same queue. And at no point should a Person who has already received 2 shots be in a queue.**

Here are the `List` interface methods you may use for a `List` where `Person` is the object type the `List` is storing. We've covered these in class, so it is expected you know what they do. You don't need to worry about import statements.

```
boolean add(Person p);
void add(int index, Person p);
boolean remove(Person p);
boolean contains(Person p);
boolean isEmpty();
Person get(int index);
int size();
```

We have also given you the Person API below. You will find the functions in this class helpful.

```
Person {
    Person(int ssn); //constructor which takes in a social security number and initializes their number of shots to be
    void vaccinate(); //method which increases the number of shots a person has received
    int getNumShots();//getter method for number of shots a person has received
    boolean equals(Object o); //determines if two person objects are equal (every person has a unique ssn identifier)
}
```

Before implementing the `VaccineQueues` class, we will write a few test cases. We highly recommend you paste the following code into your Codio environment, and then copy and paste the code for both classes in the essay textbox following this question.

Testing

The fields in `VaccineQueues` have been made public for testing convenience. You do NOT need to test for unintended side effects, just what is specified in the function headers. A test for the constructor has been completed for you as an example. Please write tests for the following cases and name them appropriately:

- A test for `joinPfizer()` where the input `Person` is already in the Pfizer queue
- A test for `joinPfizer()` where the input `Person` is not already in the Pfizer queue and has received fewer than 2 shots
- A test for `pfizerVaccinate()` where the `Person` at the head of the queue should be receiving their second shot

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

public class VaccineQueuesTest {

    private Person obj;

    // note - not necessary
    @Before
    public void setup() {
        this.queues = new VaccineQueues();
        this.obj = new Person(1);
    }

    @Test(expected=IllegalArgumentException.class)
    public void testOne() {
        queues.pfizer.add(obj);
        queues.joinPfizer(obj);
    }

    @Test
    public void testTwo() {
        queues.joinPfizer(obj);
        assertEquals(0, obj.getNumShots());
        assertTrue(queues.pfizer.contains(obj));
        assertFalse(queues.moderna.contains(obj));
    }

    @Test
    public void testThree() {
        obj.vaccinate();
        queues.joinPfizer(obj);
        queues.pfizerVaccinate();
        assertFalse(queues.pfizer.contains(obj));
        assertFalse(queues.moderna.contains(obj));
        assertEquals(1, queues.totalVaccinated);
        assertEquals(2, obj.getNumShots());
    }

}
```

Implementation

Once you have completed your tests, implement the VaccineQueues class as described. (Note: we will not ask you to implement joinModerna() and modernaVaccinate() as they are symmetric to the Pfizer functions.)

```
import java.util.*;

public class VaccineQueues {
    public int totalVaccinated; // total number of people vaccinated
    public List<Person> pfizer; // the Pfizer queue
    public List<Person> moderna; // the Moderna queue

    public VaccineQueues() {
        this.totalVaccinated = 0;
        this.pfizer = new LinkedList<>();
        this.moderna = new LinkedList<>();
    }

    public void joinPfizer(Person p) {
        boolean vaccinatedTwice = p.getNumShots() >= 2;
        boolean alreadyPresent = this.pfizer.contains(p);
        if (vaccinatedTwice || alreadyPresent) {
            throw new IllegalArgumentException("wrong");
        }
        this.pfizer.add(p);
    }

    public void pfizerVaccinate() {
        Person frontObj = this.pfizer.get(0);
        this.pfizer.remove(frontObj);
        frontObj.vaccinate();

        if (frontObj.getNumShots() < 2) {
            this.pfizer.add(frontObj);
        } else {
            this.moderna.remove(frontObj);
            this.totalVaccinated++;
        }
    }
}
```

5b: Feeding Time

During quarantine, Gian adopted dozens of cats to keep him company. He has to feed these cats 2 servings of food every night. He has both salmon and tuna food at his disposal. To prevent a feeding frenzy, Gian has trained his cats to line up in two queues, one for each type of food, each modeled as a `List` .

Each `Cat` must receive 2 servings of food - this can be both salmon, both tuna or 1 salmon + 1 tuna (the fish doesn't matter as long as they get 2 servings). With both queues, Gian gives out only 1 serving at a time. At a given point of time, a Cat can be in both queues **but cannot appear multiple times in the same queue. And at no point should a Cat who has already received 2 servings be in a queue.**

Here are the `List` interface methods you may use for a `List` where `Cat` is the object type the `List` is storing. We've covered these in class, so it is expected you know what they do. You don't need to worry about import statements.

```
boolean add(Cat c);
void add(int index, Cat c);
boolean remove(Cat c);
boolean contains(Cat c);
boolean isEmpty();
Cat get(int index);
int size();
```

We have also given you the Cat API below. You will find the functions in this class helpful.

```
Cat {

    /***
    * constructor which takes in an animal ID number and
    * initializes their servings of food to be 0
    **/
   Cat(int id);

   /***
    * method which increases the number of servings
    * a Cat has received
    **/
    void feed();

   /***
    * getter method for number of servings
    * a Cat has received
    **/
   int getNumServings();

   /***
    * determines if two Cat objects are equal
    * (every Cat has a unique animal ID identifier)
    **/
   boolean equals(Object o);
}
```

Before implementing the `FeedingQueues` class, we will write a few test cases. We highly recommend you paste the following code into your Codio environment, and then copy and paste the code for both classes in the essay textbox following this question.

Testing

The fields in `FeedingQueues` have been made public for testing convenience. You do NOT need to test for unintended side effects, just what is specified in the function headers. A test for the constructor has been completed for you as an example. Please write tests for the following cases and name them appropriately:

- A test for `joinSalmon()` where the input `Cat` is already in the Pfizer queue
- A test for `joinSalmon()` where the input `Cat` is not already in the Pfizer queue and has received fewer than 2 shots

- A test for `feedSalmon()` where the `Cat` at the head of the queue should be receiving their second shot

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

public class FeedingQueuesTest {

    private Cat obj;

    // note - not necessary
    @Before
    public void setup() {
        this.queues = new FeedingQueues();
        this.obj = new Cat(1);
    }

    @Test(expected=IllegalArgumentException.class)
    public void testOne() {
        queues.salmon.add(obj);
        queues.joinSalmon(obj);
    }

    @Test
    public void testTwo() {
        queues.joinSalmon(obj);
        assertEquals(0, obj.getNumServings());
        assertTrue(queues.salmon.contains(obj));
        assertFalse(queues.tuna.contains(obj));
    }

    @Test
    public void testThree() {
        obj.feed();
        queues.joinSalmon(obj);
        queues.feedSalmon();
        assertFalse(queues.salmon.contains(obj));
        assertFalse(queues.tuna.contains(obj));
        assertEquals(1, queues.totalFed);
        assertEquals(2, obj.getNumServings());
    }

}
```

Implementation

Once you have completed your tests, implement the `FeedingQueues` class as described. (Note: we will not ask you to implement `joinTuna()` and `feedTuna()` as they are symmetric to the Salmon functions.)

```
import java.util.*;

public class FeedingQueues {
    public int totalFed; // total number of people vaccinated
    public List<Cat> salmon; // the Pfizer queue
    public List<Cat> tuna; // the Moderna queue

    public FeedingQueues() {
        this.totalFed = 0;
        this.salmon = new LinkedList<>();
        this.tuna = new LinkedList<>();
    }


    public void joinSalmon(Cat c) {
        boolean fedTwice = c.getNumServings() >= 2;
        boolean alreadyPresent = this.salmon.contains(c);
        if (fedTwice || alreadyPresent) {
            throw new IllegalArgumentException("wrong");
        }
        this.salmon.add(c);
    }

    public void feedSalmon() {
        Cat frontObj = this.salmon.get(0);
        this.salmon.remove(frontObj);
        frontObj.feed();

        if (frontObj.getNumServings() < 2) {
            this.salmon.add(frontObj);
        } else {
            this.tuna.remove(frontObj);
            this.totalFed++;
        }
    }
}
```

5c: Pool Queues

During quarantine, Professor Harry got very good at mobile games. Specifically, he became a master at two variations of pool: eight-ball and nine-ball. Some of the students heard about Harry's skills and wanted to challenge him. Harry told them to make two pool queues so he could orderly play all the challengers, one for eight-ball games and one for nine-ball games, with each modeled as a List. He decided he would play each student exactly twice.

Each Student must play Harry 2 times - this can be both eight-ball games, both nine-ball games, or 1 of each. With both queues, Professor Harry plays only one game at a time. At a given point of time, a Student can be in both queues **but cannot appear multiple times in the same queue. And at no point should a Student who has already played 2 games be in a queue.**

Here are the `List` interface methods you may use for a `List` where `Student` is the object type the `List` is storing. We've covered these in class, so it is expected you know what they do. You don't need to worry about import statements.

```
boolean add(Student s);
void add(int index, Student s);
boolean remove(Student s);
boolean contains(Student s);
boolean isEmpty();
Student get(int index);
int size();
```

We have also given you the `Student` API below. You will find the functions in this class helpful.

```
Student {
    /***
     * constructor which takes in an ID number and
     * initializes their games played to be 0
     **/
    Student(int id);

    /***
     * method which increases the number of
     * games a Student has played
     **/
    void play();

    /***
     * getter method for number of games
     * a student has played
     **/
    int getNumGamesPlayed();

    /***
     * determines if two Student objects are equal
     * (every student has a unique ID number)
     **/
    boolean equals(Object o);
}
```

Before implementing the `PoolQueues` class, we will write a few test cases. We highly recommend you paste the following code into your Codio environment, and then copy and past the code for both classes in the essay textbox following this question.

Testing

The fields in `PoolQueues` have been made public for testing convenience. You do NOT need to test for unintended side effects, just what is specified in the function headers. A test for the constructor has been completed for you as an example. Please write tests for the following cases and name them appropriately:

- A test for `joinEightBall()` where the input `Student` is already in the eight-ball queue
- A test for `joinEightBall()` where the input `Student` is not already in the eight-ball queue and has played fewer than 2 games
- A test for `playEightBall()` where the `Student` at the head of the queue should be playing their second game

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

public class PoolQueuesTest {

    @Test
    public void constructorTest() {
        PoolQueues q = new PoolQueues();
        assertTrue(q.eightBall.isEmpty());
        assertTrue(q.nineBall.isEmpty());
        assertEquals(0, q.totalPlayed);
    }

    @Test(expected=IllegalArgumentException.class)
    public void testOne() {
        queues.eightBall.add(obj);
        queues.joinEightBall(obj);
    }

    @Test
    public void testTwo() {
        queues.joinSalmon(obj);
        assertEquals(0, obj.getNumGamesPlayed());
        assertTrue(queues.eightBall.contains(obj));
        assertFalse(queues.nineBall.contains(obj));
    }

    @Test
    public void testThree() {
        obj.play() = 1;
        queues.joinEightBall(obj);
        queues.playEightBall();
        assertFalse(queues.eightBall.contains(obj));
        assertFalse(queues.nineBall.contains(obj));
        assertEquals(1, queues.totalPlayed);
        assertEquals(2, obj.getNumGamesPlayed());
    }


}
```

Implementation

Once you have completed your tests, implement the PoolQueues class as described. (Note: we will not ask you to implement joinNineBall() and playNineBall() as they are symmetric to the eight-ball functions.)

```java
import java.util.*;

public class PoolQueues {
    public int totalPlayed; // total number of Students played
    public List<Student> eightBall; // the eight-ball queue
    public List<Student> nineBall; // the nine-ball queue

    public PoolQueues() {
        this.totalPlayed = 0;
        this.eightBall = new LinkedList<>();
        this.nineBall = new LinkedList<>();
    }

    public void joinEightBall(Student s) {
        boolean playedTwice = s.getNumGamesPlayed() >= 2;
        boolean alreadyPresent = this.eightBall.contains(s);
        if (playedTwice || alreadyPresent) {
            throw new IllegalArgumentException("wrong");
        }
        this.eightBall.add(s);
    }

    public void playEightBall() {
        Student frontObj = this.eightBall.get(0);
        this.eightBall.remove(frontObj);
        frontObj.play();

        if (frontObj.getNumGamesPlayed() < 2) {
            this.eightBall.add(frontObj);
        } else {
            this.nineBall.remove(frontObj);
            this.totalPlayed++;
        }
    }
}
```