CS212: Paradigms
Dr. Gross, Spring 2016

Project 2: Would You Like to Play a Game of Roshambo? 

1. In your **homework repo**, create a new package called `edu.blackburn.cs.cs212sp16.lastname.airoshambo`
2. Copy the file airoshambo-shell.uml into this directory and rename airoshambo-lastname.uml

Let's face it, roshambo is kinda boring. It's not a game you can be good at… or is it? If human beings were truly random, then no, but we're not even remotely random. We think we are, but we're not. Let's sprinkle some AI magic and see if we can make a computer player that performs better than average. In order to do that, we need to start with a solid program that follows good OO design. *All submissions due before class that day.*

Understanding Enumerations

In order to complete this assignment, we're going to use enumerations. Enumerations (enums) are kinds of classes in that they are subclasses of Object, but all they really are is a list of values. For example, a BlackburnRole enum might have the values STUDENT, STAFF, and FACULTY. Here's the code:

```
public enum BlackburnRole {
    STUDENT, STAFF, FACULTY
}
```

You use an enum as a datatype, just as you would other classes. However, you don't have to use new to create instances, because the instances already exist, and are named the values. Here's how the enum can be used:

```
BlackburnRole p1 = BlackburnRole .STUDENT;
BlackburnRole p2 = BlackburnRole .STAFF;
BlackburnRole p3 = BlackburnRole .FACULTY;
```

We could also have a Person class with an attribute called role of type BlackburnRole , so:

```
public class Person {
      private String name;
      private BlackburnRole role;
      public Person(String name, BlackburnRole role) {
            This.name = name;
            This.role = role;
      }

      public BlackburnRole getRole() {
            return this.role;
      }

}
```

I've given you code in the Project2 folder for the Move enum, along with code that uses it, but you'll need to write your own Winner enum.

Part i: Initial Design (10/100, due Mon, Apr 11)

I've given you a shell class diagram with important classes and a method or two. In your copy of the class diagram, add in any classes, attributes (which should be private, but imply a setter and/or getter), and methods that you will need. Arguments and return types should be specified. Your design should handle:

- A game of roshambo with
  - A human player, who chooses their moves
  - A computer player, who decides moves based on some algorithm
  - A set of statistics for a game
- At the beginning of each round, each player is prompted to choose their move
  - The play() method should return the move
  - The round should be recorded, a winner (or tie) determined, and added to the statistics
  - The round should be passed back to each player
- You need to have one subclass of ComputerPlayer called RandomPlayer
  - It will randomly select a move
  - You need to figure out how to get from a random number to a Move enum element

The class diagram should keep you from wandering too far from the easiest path. Huh. The human player's move. That's funny. The superclass has a move() method, but that means that nothing is passed into the object. How will you get a move from the user into the method? This would be a good question for an expert. Or Dr. Gross. Whichever is available. The solution is something you know, but is not obvious.

Part $\sqrt{2}$: Implementing the Game (15/100, due Thursday, Apr 14)

OK, now you need to program this puppy. No UI yet. Start with the classes and methods. Get them working. Some of this will be based on your code from last week. Start with hardcoded (or random) moves for the user. Once that's working, you can use Scanner to get input if you want. You should have no more than five lines of code in you main() method. We're doing OO here.

Part e: Building a UI Around the Game (10/100, due Monday, Apr 18)

With a working game *now you can build the UI*, by hand or with FXML. It should not be fancy. All you need is:

- The number of the most recent round
- The win count and win percent for each player
- The tie count and tie percent
- Each player's move for the most recent round
- The outcome of that round

The only controls you will need are buttons for rock, paper, and scissors, and an exit button would be nice. Only one window. Can you add more GUI features? Yes, but later.

Part ΠΤ: The Circular Linked List & First AI Method (65/100, due Thu, Apr 21)

The circular linked list (CLL) is just a linked list where the last element points to the first element. As a result, there's no last element or first element. In this case, the linked list should be specialized to:

- Keep track of Round objects
- Internally determine the next move for the player
- Have an add() method that either
  - Adds the Round object (if the CLL is not at 30 entries)
  - Replaces the oldest element in the
- How will you loop through if there's no beginning and no end? Hints:
  - You'll use a do-while loop
  - You'll need to know what gets compared if you compare two objects with ==

First, create another subclass of ComputerPlayer called LruPlayer. It's the only class that needs to know anything about the CLL. Then create a subclass of CircularLinkedList called LruCll.

Least Recently Used (LRU) Algorithm

Since your CLL will contain the recent games, determine which move has been used the least in those games, based on the assumption that this one will be more likely to be used. Create a LruCll class. When asked to recommend a move, it should find the move in the CLL used the least and return that. In the case of a tie (or an empty array), return a move randomly. If the tie is only between two moves, choose one of those moves.

Extra Credit

You can do none, any, or all of these for extra credit.

Multiplayer Mode

- Have multiple games running at any given time
- You already have at least two subclasses of ComputerPlayer: LruPlayer and RandomPlayer
  - Create one instance of HumanPlayer
  - Create one instance of each subclass of ComputerPlayer
- A player should be able to play against multiple players simultaneously
  - There need to be separate methods for choosing a move and returning the move
  - Each time that player chooses a move, they are choosing a new move
  - They should return that move until told to move again
  - This will require some minor changes to Player and all subclasses
- Each pair of human and computer player will have its own game and stats
  - Have each player (including the human) each make one move
  - Compare the one human player move to each computer player move
  - Have a separate pane (on the same window) for the outcome and stats for each pair

Weighted LRU Algorithm

This will be an extension of the LRU approach (subclass, hint, hint?). In this case, when asked to play, the Weighted LRU will average the distance for each move from the most recent move. To do this, it will need a count and sum for each kind of move. When it sees a move, it should add one to that move's counter and add the distance from the current move to that move's sum. For example, if the twentieth most recent move was

rock, the player would add one to the rock count and 20 to the rock sum. When it has analyzed all of the moves, it should get the average distance for each move from the current move by dividing move count by move sum; whichever move average is largest should be played.

Last Move Sequence Algorithm

The LastMoveSequenceCll will need to remember the last move the opponent played. It will then analyze the CLL for the most frequently played move after that move. For example, if the last move was Rock, and of the last thirty plays, the most common move after Rock was Paper, this CLL should suggest Paper.

Last N Moves Sequence Algorithm

The LastNMovesSequenceCll will need extend the LastMoveSequenceCll by looking at two or more moves. So if the opponent's last two moves were Rock then Scissors, you would look back in the CLL for the most common move after that pair. There are only nine possible pairs, so hopefully you'll have multiple instances in the CLL for this player to find. If it can't find any, it should fall back on LastMoveSequence.

Skynet

Let's take the human out of the loop. Just have the computer players play against each other. Wow, they work fast. How do you show what's happening? Do you need to show the outcome of each round? If so, is there a better way than very, very briefly flashing that outcome on the screen, too fast to read?

Nearly Headless

Forget the whole UI. Let's just have computer players run. Print out stats to a file at the end of a run. Raise the size limits of the CLLs. Does that change performance?

Even More Nearly Headless

Multithread this so that each player is running in its own thread. Have ComputerPlayer implement Runnable. Oh, wait, if each player is in its own thread, will that have any effect if the games are run in series? (No.) So make game implement Runnable. Wait, Game objects and ComputerPlayer objects running in their own threads? Yes. Won't that be a problem, because how will I know when it's OK to change each player's move? Yes. How do I fix that? Use the methods wait() and notifyAll(), both on all objects.

And Now for Something Completely Different

Make your own AI algorithm. Think about how to make a computer, which is good at memorizing and doing math, think through the problem and suggest a good move.

But I Really Like GUI Programming

OK, that's fine. There's plenty of room for a better GUI. Better layout. Better design. Animation. Colors. Lasers. Frickin laser beams attached to their heads. You can do a lot with JavaFX. One possibility is adding graphs of the data you are collecting. But c'mon, that's boring. Graphs. Pfft.

Final Thoughts

This is the *last Java program of your first year in Computer Science*. In fact, it's the *last Java program you'll write for a class in 2016*. So yeah, it's a bit hard. Ihave:

- Give you reasonable hints
- Point out the sticky bits
- Break the work into manageable chunks

I've moved a lot to extra credit because I would rather you focus on the core than try to get parts done. That said, this is something that would be a good project to extend *over the summer*.

Rubric

Design (Class Diagram)	**10**
Design	5
Time estimate for each part (add to a text file in package)	5
Implementing Game	**25**
Implementation matches design	5
Correctly capturing/reporting statistics	10
Five or fewer lines in main()	5
Comments	5
UI	**10**
Working UI	5
Commented controller	5
Final	**55**
Standards/comments	10
Time estimate/accounts	10
CLL works	15
Properly recommend via LRU algorithm	15
Asking me a full question in office hours or via email	5
Total	**100**