Assignment 4

Due Feb 25

1. This problem does **NOT** involve inheritance. It uses interacting classes with "has-a" relationships. This is the most complex program so far. It will take some design.

Write a program which will allow a person to play the slot machine pictured below.

Here's how the slot machine works:

A slot machine game works as follows:

A player can insert any number of coins (say dollar coins) into the machine. Once the there is money in the machine, the player can make a bet. The player can bet up to three coins. If he/she pushes the BetMax Button, the bet is three coins and 3 is subtracted from the current cash that is in the machine. Each time he/she pushes the BetOne button, the bet is upped by one coin. (The maximum bet is three coins , however). Notice that the BetOne button can be used to bet two coins-- push it twice. When the player pushes the Spin Button, the wheels turn and the game is played. If the player wins, coins are added to the cash in the machine; if the player loses nothing is added (since the bet has already been subtracted from cash). The player can get all of his/her money by pushing the CoinOut button.

The winning symbols on the machine are:

- Three Sevens in each window (i.e. in all three windows)
- One Seven in each window
- Three Bars in each window
- Two Bars in each window
- One Bar in each window
- Three Cherries, one in each window
- Two Cherries, one cherry in any two windows
- One Cherry in any window

The payoff is as pictured on the machine.

(Note: the numbers in column one are 200, 50,30,20,15,10, 5,and 2. The other columns are two and three times column one

Notice that the payoffs depends on the number of coins bet. So for example, if you spin

with a bet of one coin, the payoff is 200
with a bet of two coins, the payoff is 400 and
with a bet of three coins, the payoff is 600.
(The amount you actually win is 199, 498, 597 -- since your bet was already collected.)

Note : There is a payoff for one cherry in any position, and two cherries in any positions.

Designing the program:

The first step is to decide what classes/objects you will need. Re-read the problem statement. From the specification of the problem , it is clear that there are three major entities: a player, a slot machine, and a game . Each of these entities will be an object in the program. Moreover, these objects interact with each other. (Note: The nouns of the problem statement-- *game, person, slot machine* -- can give you an idea of what objects you might use.)

Next, design the classes -- a SlotMachine class, a Player class, and a Game . Let's start with the slot machine. What are the actions of a slot machine? The behaviors? The methods? What can you do with a slot machine? What is the interface? If you look at the picture. You can see that there are a number of things a slot machine can do (or you can do with a slot machine):

- Bet one coin (BetOne button, i.e. void pushBetOne())
- Bet three coins (BetMax button)
- Play the game (Spin button)
- Return cash to player (CoinOut button)
- Accept money (Inset Coin slot)

For now, these functions will serve as the public interface of the slot machine. These are the methods.

We now must decide what kind of data do we need to describe the state of the slot machine? What is "inside" the machine? What is the "private data?"

- The number of coins currently in the machine
- The current bet

You may think of other data or methods for your class. That's OK.

The next step is to design the SlotMachine class and make sure that it works correctly.

In your design, you must be sure that you catch all errors that a player might make. Some possible errors are:

- a player pushes spin before making a bet-- can't play without betting
- a player bets more than the cash he/she has left in the machine. If the machine has only 2 coins, the player cannot bet 3 before inserting more money.
- a player tries to bet more than 3 coins. The maximum bet is three.

It's time to build the class. Think about what each method must do and how each method alters the state of the machine (i.e. the data). For example,

- the method pushBetOne() will increment the current bet by 1 (coin) and also decrement the number of coins by 1. (Of course, there must be at least one coin in the machine)
- The method insertCoins(int n) will add n coins to the cash .

Each method will change the state of the machine. Obviously, the method pushSpin() will do a good deal of the work.

Because we do not (yet) have graphics capabilities, for our output we'll let

- 777 denote Triple Seven (So 777 777 777 is the jackpot)
- 7 denote Seven (7 7 7 pays 50 on a one coin bet)
- $$$ denote Triple Bar ($$$ $$$ $$$ pays 30 on a one coin be)
- $$ Denote Double Bar ($$ $$ $$ pays 20 on a one coin be)

- $ Denote Single Bar ($ $ $ pays 15 on a one coin be)
- @ denote Cherry

There is one additional design issue-- an issue very important to casinos: how frequently will the various pictures appear -- how often will the machine make a payout? It is not totally random.

- Should 777 777 777 (jackpot) turn up as frequently as @ @ @? Probably not.
- Should $$$ $$$ $$$ appear less frequently than $ $ $?

Suggestion: To program the machine, generate three random integers in the range of, say, 0..17. If a random number is

- 0 -- display 777
- 1 or 2 -- display a 7
- 3, 4, or 5 -- display $$$
- 6, 7,8, or 9 -- $$
- 10,11,12,13, or 14 -- $
- 15, 16, or 17 -- @ (cherry)

Thus if the three random numbers are 3 15 and 2 the display is

$$$ @ 7 (no payoff)

The slot machine makes a payoff about 46% of the time. Actually this is a pretty cheap machine. Machines in Vegas pay much more!

Notice that the combination 777 777 777 (for example) will be pretty rare. In fact, it will occur only one time in about 5800 plays but @ @ @ will appear about once in every 216 spins.

Ok, now you should implement and test your slot machine class. Make sure you add enough output statements to your code so that you can see what is happening. You can remove extra output statements after your class is debugged.

Add a main method to test all the buttons of the machine. When you are convinced the machine works, remove the main method.

Part II: The Player Class.

Like a slot machine object we need to determine what a player object can do. The player object will interact with a slot machine object Some of the things a player object may do are:

- push the BetOne button of a machine (public void betOne())
- push the BetMax button (
- push the CoinsOut button
- push the Spin button
- insert coins into the machine
- see how much money is left in his/her wallet(int getbankroll())
- get some more money -- maybe from the ATM.

What data describes the state of a player object? What is the private data? To play the game a player must be in poession of a SlotMachine object...

public class Player---
  private SlotMachine m
  public methods:
    player(); // default constructor, instantiates the SlotMachine object
    void pushBetOne();
    void pushBetMax();
    void pushCoinsOut();
    void pushSpin();
    void insertcoins();
    void exitTheGame()

Implement and test a player object.

Part II. The Game

The Game class has a player object
    private Player player;

There should also be a play() method that

1. presents a menu to the player

2. carries out the choice of the player

Here is a picture of typical output:

```
Choose one of the following
1: Push BetOne
2: Push BetMax
3: Push Spin
4: Insert Coins
5: Push CoinOut
6: Exit
? 1

Not enough cash. Deposit coins first

Choose one of the following
1: Push BetOne
2: Push BetMax
3: Push Spin
4: Insert Coins
5: Push CoinOut
6: Exit
? 4
How many coins? 20

$20 inserted. Cash in machine is $20


Choose one of the following
1: Push BetOne
2: Push BetMax
3: Push Spin
4: Insert Coins
5: Push CoinOut
6: Exit
? 3

Must make bet before spinning

Choose one of the following
1: Push BetOne
2: Push BetMax
3: Push Spin
4: Insert Coins
```

```
5: Push CoinOut
6: Exit
? 2

Bet is now $3


Choose one of the following
1: Push BetOne
2: Push BetMax
3: Push Spin
4: Insert Coins
5: Push CoinOut
6: Exit
? 3


S L O T M A C H I N E

@ $ 7

BET: 3
PAYOUT: 6
CASH REMAINING IN MACHINE: 23



Choose one of the following
1: Push BetOne
2: Push BetMax
3: Push Spin
4: Insert Coins
5: Push CoinOut
6: Exit
? 5

Returning $23


Choose one of the following
1: Push BetOne
2: Push BetMax
3: Push Spin
4: Insert Coins
5: Push CoinOut
6: Exit
```

?

Once these three classes are working, make a very small class

```
public class PlaySlotMachine
{
    public static void main(String[] args)
    {
        Game g = new SlotGame();
        g.play();
    }

}
```

Notice that main(...) is very simple. All the functionality is in the three classes.

Design Question: When a player exits, what happens to the cash in the machine? Is it automatically returned or does the player have to press CoinOut to get what is left? I programmed it so that the player has to press CoinsOut. If he/she forgets-- well the money stays in the machine--stupid player. You can do it either way but for consistency in grading, assume that the player must push the coins out button to get money out of the machine. Exit means "just walk away."

Problem 2. This problem exploits inheritance

There are many different types of lists into which data may be inserted and removed.

In this problem you will implement three types of lists:
A LIFO list, a FIFO list, and a PRIORITY list

Each of these list have similar but different methods

```
insert(x) // inserts x into the list
remove() // removes and returns an item from the list
```

as well as a common method

```
getSize() // returns the number of items in the list
```

****

A **LIFO list** is a "Last In -- First Out" list.

So

```
insert(x) places x at the end of the list
remove() removes and returns the last item in the list
```

For example, if a is a LIFO list the code

```
a.insert(4);
a.insert(7);
a.insert(3);
```

produces a list of size 3 that looks like

4 7 3 // the three was the last one put into the list

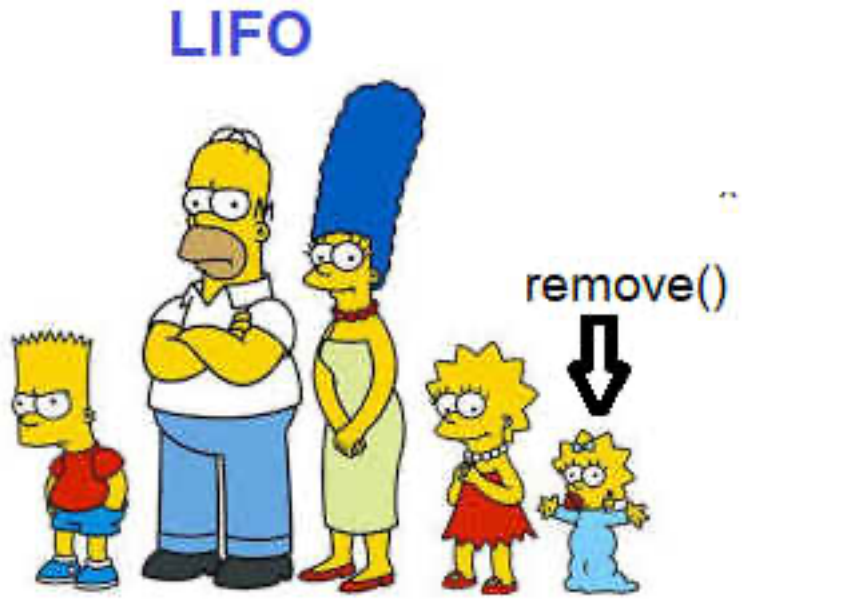
and the code

```
System.out.println(a.remove());
 System.out.println(a.remove());
System.out.println(a.remove());
```

produces output

```
3
 7
 4
```

Last value inserted is the first value removed.



LIFO --Last in First Out

Maggie was the last to get in line, so the first to be removed

****

A **FIFO list** is a "First in - First Out" list. A FIFO list is like an ordinary waiting line. The first person in line is the first person served.

For example, if b is a FIFO list the cose

```
b.insert(4);
b.insert(7);
b.insert(3);
```

produces a list of size 2 that looks like

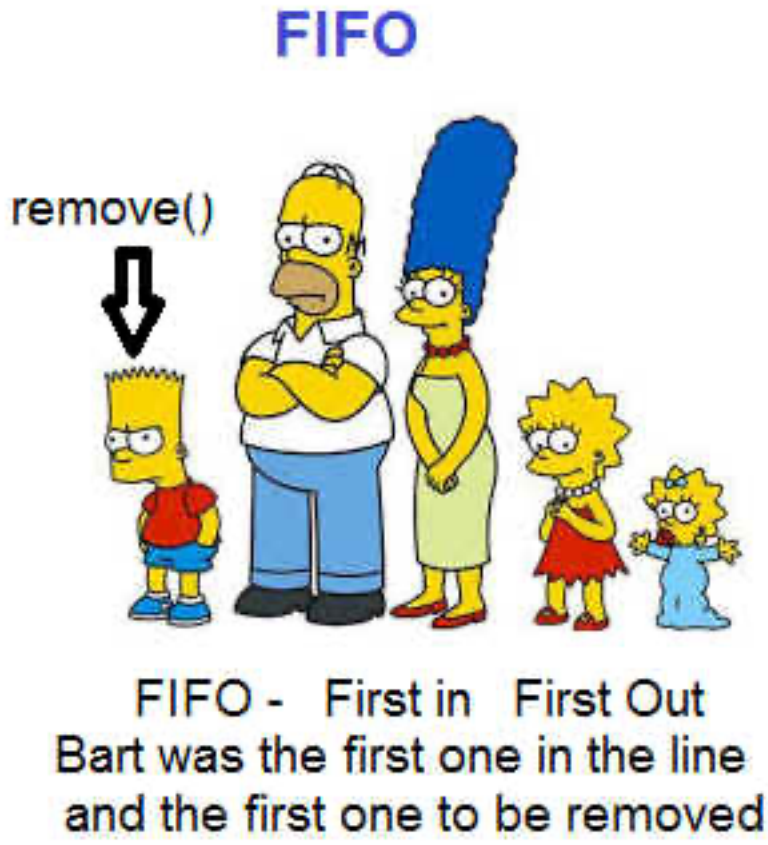
4 7 3

and the code

```
System.out.println(b.remove());
System.out.println(b.remove());
System.out.println(b.remove());
```

produces output

```
4
  7
      3
```



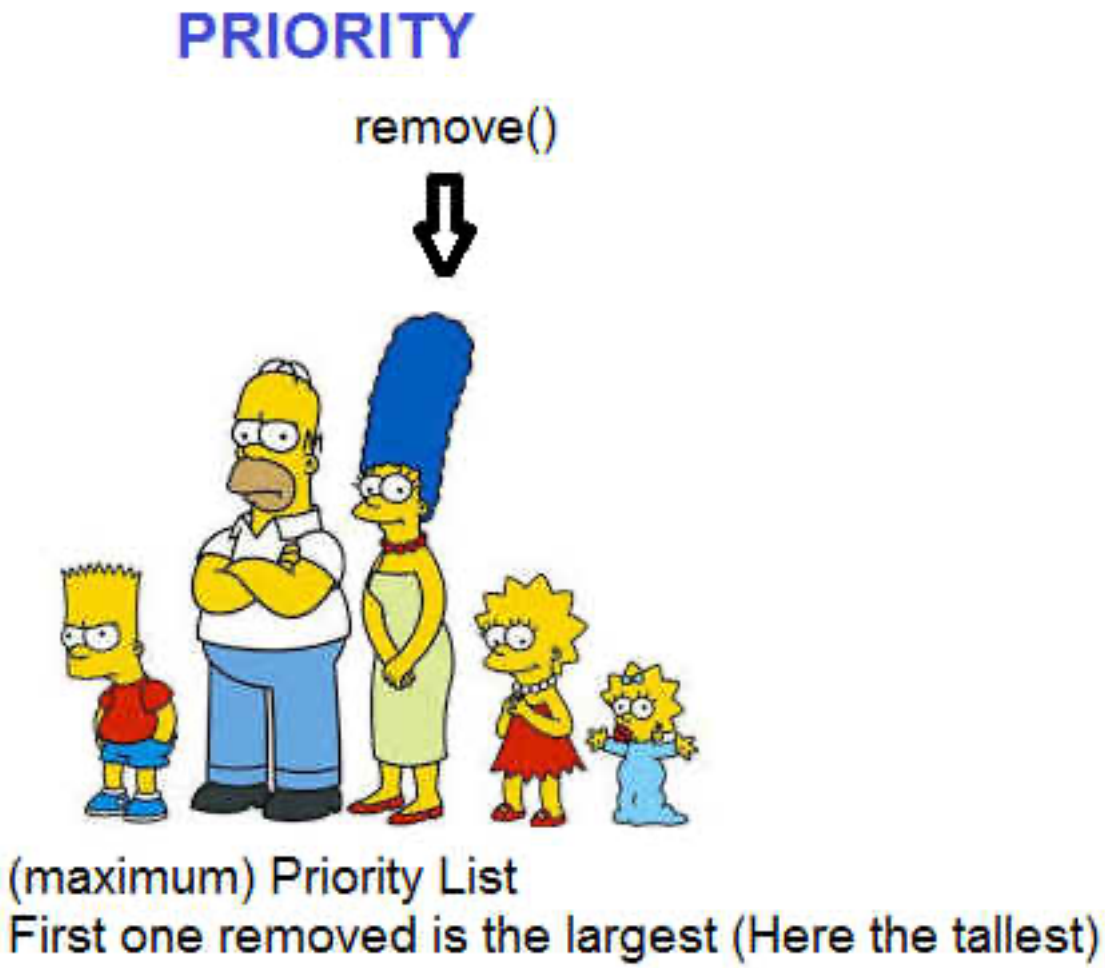
****

With a **(Maximum) PRIORITY** list a call to remove() removes and returns the item of highest priority. If the list is a list of integers, the remove() removes and returns the maximum integer.

For example, if c is an PRIORITY list the code

```
c.insert(4);
        b.insert(7);
   b.insert(3);
```

remove() removes and returns 7.



Notice that each list has its own insert and remove method.

For this assignment, assume all lists are lists of int.

1. **Implement an abstract class Lists:**

- Use and array (int [] data) NOT an ArrayList to store numbers in the list and
- Include a variable size to keep track of the number of items in the list.
- Assume a list is no larger than 100.
- Let default constructor create an array of size 100 and set size to 0
- Include abstract methods

i. int remove()

ii. void insert(int x) and

implement the method

int getSize()

Now make three classes: LIFO, FIFO, and PRIORITY that implement the three types of list mentioned above.

LIFO LISTS:

Insertion and removal from the LIFO list is easy. Everything is done at the end.

FIFO LISTS:

For the FIFO list, insert data at the end and remove the data at position 0.

Each time you remove a number from the list shift the remaining elements "to the left."

For example if the list is

7	4	9	1	2	3	8													
0	1	2	3	4	5	6	...	...											

a call to remove() removes and returns 7 and shifts all the data {"to the left") so that the 4 is now in position 0:

4	9	1	2	3	8													
0	1	2	3	4	5	...	...											

(size is now 6)

(There are more efficient ways to implement a FIFO list and we will see them later)

PRIORITY LIST:

For the PRIORITY list, when inserting a number x, just place x at the end of the list, as you did with the LIFO list.

DO NOT SORT THE LIST AFTER EVERY INSERTION.

To remove a value

1. search the list for the maximum value as well as its position, maxPosition

2. shift the data in the array starting at maxPosition+1;

3. return the highest value

So suppose the list is

7	4	9	1	2	3	8												
0	1	2	3	4	5	6	...	...										

size is 7

remove() -- the maximum is 9, in position 2. Remove (and return the 9)

Shift the values following the 9 (from position 3) "to the left." (Those are the green numbers , 1,2,3,and 8.)

7	4	1	2	3	8													
0	1	2	3	4	5	6	...	...										

The value 9 has been removed, values 1,2,3,and 8 have been shifted and size is now 6

NOTE:
For all lists, each time you remove an item, you must decrement *size* and each time you insert an item you must increment *size.*

If the list is empty (i.e. size ==0), a call to remove() is obviously an error. In that case, print that message "Empty List" and return (and print) the value -999.

Test your List hierarchy with the following class:
(Don't type it...cut and paste it into your program.)

```
public  class TestLists
{
   public static void main(String[] args)
   {
      LIFO s = new LIFO();
      System.out.println("LIFO: ");
      s.insert(2);
      s.insert(12);
      s.insert(71);
      s.insert(50);
      System.out.println(s.remove());
      System.out.println(s.remove());
      s.insert(3);
      s.insert(13);
      System.out.println(s.remove());
      System.out.println(s.remove());
      s.insert(11);
      System.out.println(s.remove());
      System.out.println(s.remove());
      System.out.println(s.remove());
      System.out.println(s.remove());

       FIFO q = new FIFO();
      System.out.println("FIFO: ");
      q.insert(2);
      q.insert(12);
      q.insert(71);
      q.insert(50);
      System.out.println(q.remove());
      System.out.println(q.remove());
      q.insert(3);
      q.insert(13);
      System.out.println(q.remove());
      System.out.println(q.remove());
      q.insert(11);
      System.out.println(q.remove());
      System.out.println(q.remove());
```

```
        System.out.println(q.remove());
        System.out.println(q.remove());

        PRIORITY pq = new PRIORITY();
         pq.insert(2);
        pq.insert(12);
        pq.insert(71);
        pq.insert(50);
        System.out.println(pq.remove());
        System.out.println(pq.remove());
        pq.insert(3);
        pq.insert(13);
        System.out.println(pq.remove());
        System.out.println(pq.remove());
        pq.insert(11);
        System.out.println(pq.remove());
        System.out.println(pq.remove());
        System.out.println(pq.remove());
        System.out.println(pq.remove());
    }
}
```

.