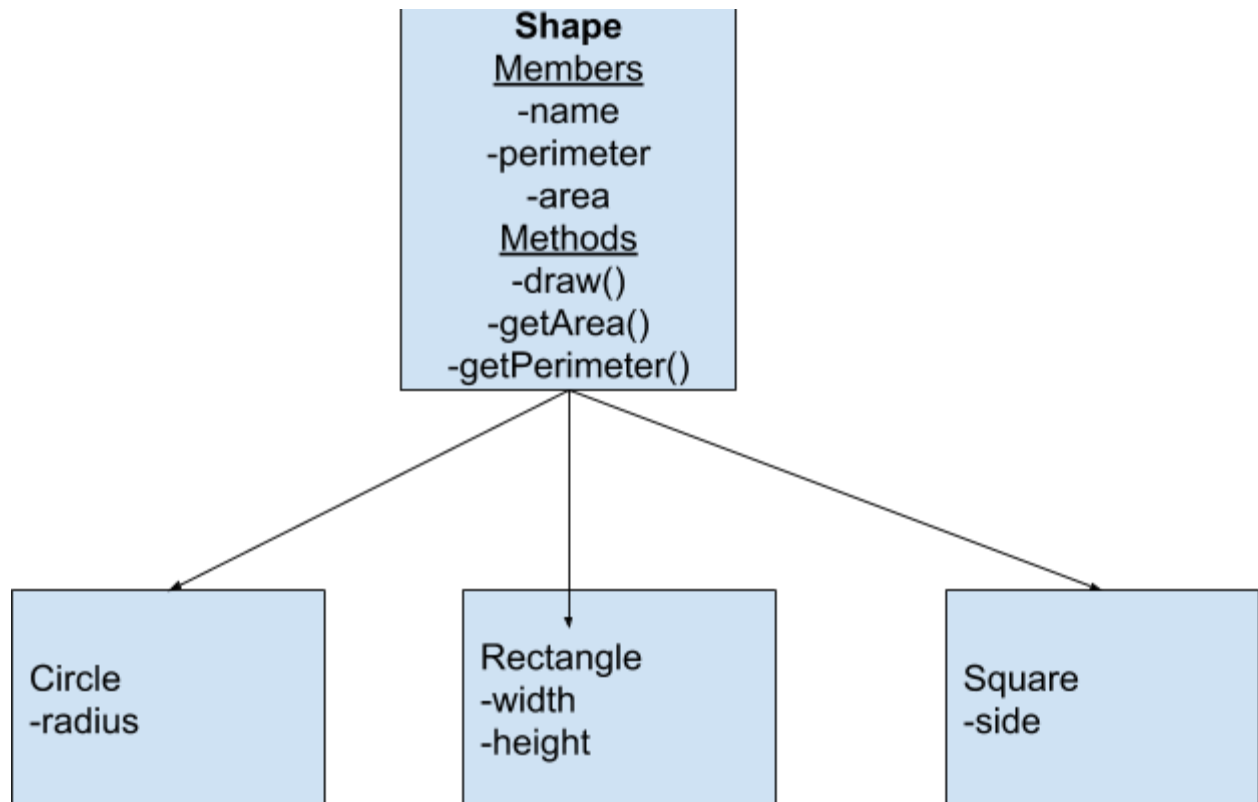


**Deadline: 7th October 2019, 11:59pm**

**Q1.** Implement the following four classes using Inheritance (2 mks)



Use the following class to test your classes.

```
public class ShapeTester
{
    //Main operation class for testing
    public static void main(String args[])
    {
        Shape s = new Shape();
        Rectangle r = new Rectangle(2.0, 3.0);
        Circle c = new Circle(4.0);
        Square sq = new Square(4.0);

        r.getArea(); // should print 6.0
        r.getPerimeter(); // should print 10.0
        c.getArea(); // should print 50.26
        c.getPerimeter(); // should print 25.13
        sq.getArea(); // should print 16.0
        sq.getPerimeter(); // should print 16.0
        r.draw(); // should print "Drawing rectangle"
```

```

    c.draw(); // should print "Drawing Circle"
    s.draw(); // should print "Drawing Shape"
    sq.draw(); // should print "Drawing Square"
}
}

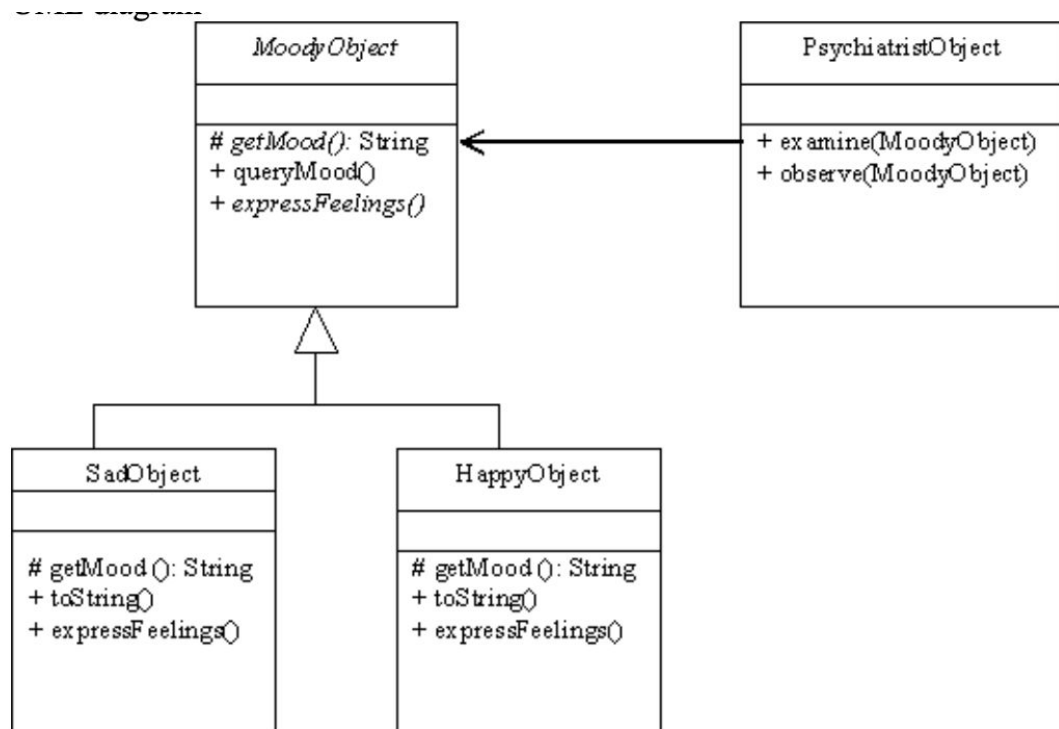
```

Q2. For this problem you will write a Java program that uses polymorphism

and abstract classes and methods. The program should implement the design indicated

in this UML diagram

(3mks)



**MoodyObject:**

//returns the mood: sad or happy - depending on which object sends the message

protected abstract String getMood();

//each object expresses a different emotion

```
protected abstract void expressFeelings();
```

```
//an object responds according to how it feels, print "I feel happy(or sad) today!"
```

```
public void queryMood() {
```

**SadObject:**

```
//returns a string indicating sad
```

```
protected String getMood();
```

```
//print crying string: " 'wah' 'boo hoo' 'weep' 'sob' 'weep' "
```

```
public void expressFeelings();
```

```
//returns message about self: "Subject cries a lot";
```

```
public String toString();
```

**HappyObject:**

```
//returns a string indicating happy
```

```
protected String getMood();
```

```
//print laughter string: "hehehe...hahahah...HAHAHAHAHA!!!"
```

```
public void expressFeelings();
```

```
//returns message about self: "Subject laughs a lot"
```

```
public String toString();
```

### PsychiatristObject:

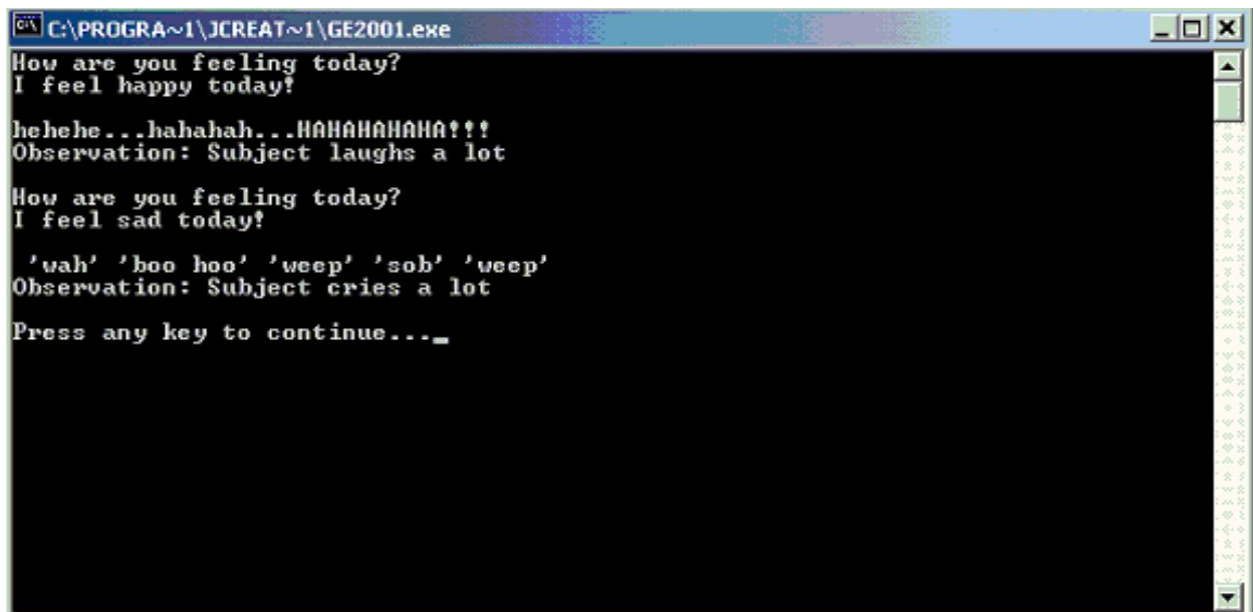
//asks a moody object about its mood

```
public void examine(MoodyObject moodyObject);
```

//a moodyObject is observed to either laugh or cry

```
public void observe(MoodyObject moodyObject);
```

Write a main() method that creates a psychiatrist object and two moodyObjects. The psychiatrist object will examine and observe each moodyObject. The output of your program should be same with the following output:



```
C:\PROGRA~1\JCREAT~1\GE2001.exe
How are you feeling today?
I feel happy today!
hehehe...hahahah...HAHAHAHAHA!!
Observation: Subject laughs a lot
How are you feeling today?
I feel sad today!
'wah' 'boo hoo' 'weep' 'sob' 'weep'
Observation: Subject cries a lot
Press any key to continue..._
```

Q3. (4 mks) You will be writing code in support of a DessertShop which sells candy by the pound, cookies by the dozen, ice cream, and sundaes (ice cream with a topping). Your software will be used for the checkout system.

To do this, you will implement an inheritance hierarchy of classes derived from a `DessertItem` abstract superclass.

The `Candy`, `Cookie`, and `IceCream` classes will be derived from the `DessertItem` class.

The `Sundae` class will be derived from the `IceCream` class.

You will also write a `Checkout` class which maintains a list (`Vector`) of `DessertItems`.

## The `DessertItem` Class

The `DessertItem` class is an abstract superclass from which specific types of `DessertItems` can be derived. It contains only one data member, a name. It also defines a number of methods. All of the `DessertItem` class methods except the `getCost()` method are defined in a generic way in the file `Class-DessertItem`, provided for you along with the other homework specific files in the directory. The `getCost()` method is an abstract method that is not defined in the `DessertItem` class because the method of determining the costs varies based on the type of item. Tax amounts should be rounded to the nearest cent. For example, calculating the tax on a food item with a cost of 199 cents with a tax rate of 2.0% should be 4 cents. The complete specifications for the `DessertItem` class are provided in the file `Class-DessertItem`.

## The `DessertShop` Class

The `DessertShop` class contains constants such as the tax rate as well as the name of the store, the maximum size of an item name and the width used to display the costs of the items on the receipt. Your code should use these constants wherever necessary! The `DessertShop` class also contains the *`cents2dollarsAndCents`* method which takes an integer number of cents and returns it as a `String` formatted in dollars and cents. For example, 105 cents would be returned as "1.05".

## The Derived Classes

All of the classes which are derived from the `DessertItem` class must define a constructor. The `TestCheckout` class determine the parameters for the various constructors. Each derived class should be implemented by creating a file with the correct name, eg., `Candy.java`.

The Candy class should be derived from the DessertItem class. A Candy item has a *weight* and a *price per pound* which are used to determine its *cost*. For example, 2.30 lbs. of fudge @ .89 /lb. = 205 cents. The cost should be rounded to the nearest cent.

The Cookie class should be derived from the DessertItem class. A Cookie item has a *number* and a *price per dozen* which are used to determine its *cost*. For example, 4 cookies @ 399 cents /dz. = 133 cents. The cost should be rounded to the nearest cent.

The IceCream class should be derived from the DessertItem class. An IceCream item simply has a *cost*.

The Sundae class should be derived from the IceCream class. The *cost* of a Sundae is the *cost* of the IceCream plus the *cost of the topping*.

## The Checkout Class

The Checkout class provides methods to enter dessert items into the cash register, clear the cash register, get the number of items, get the total cost of the items (before tax), get the total tax for the items, and get a String representing a receipt for the dessert items. The Checkout class must use a Vector to store the DessertItems. The total tax should be rounded to the nearest cent. The complete specifications for the Checkout class are provided in the file Class-Checkout.

## Testing

A simple testdriver, TestCheckout.java along with its expected output, are provided for you to test your class implementations. You can add additional tests to the driver to more thoroughly test your code.

## TestCheckout

```
public class TestCheckout {  
  
    public static void main(String[] args) {  
  
        Checkout checkout = new Checkout();  
  
        checkout.enterItem(new Candy("Peanut Butter Fudge", 2.25, 399));  
        checkout.enterItem(new IceCream("Vanilla Ice Cream", 105));  
        checkout.enterItem(new Sundae("Choc. Chip Ice Cream", 145, "Hot Fudge",  
50));  
        checkout.enterItem(new Cookie("Oatmeal Raisin Cookies", 4, 399));  
    }  
}
```

```

        System.out.println("\nNumber of items: " + checkout.numberOfItems() +
"\n");
        System.out.println("\nTotal cost: " + checkout.totalCost() + "\n");
        System.out.println("\nTotal tax: " + checkout.totalTax() + "\n");
        System.out.println("\nCost + Tax: " + (checkout.totalCost() +
checkout.totalTax()) + "\n");
        System.out.println(checkout);

        checkout.clear();

        checkout.enterItem(new IceCream("Strawberry Ice Cream", 145));
        checkout.enterItem(new Sundae("Vanilla Ice Cream", 105, "Caramel",
50));
        checkout.enterItem(new Candy("Gummy Worms", 1.33, 89));
        checkout.enterItem(new Cookie("Chocolate Chip Cookies", 4, 399));
        checkout.enterItem(new Candy("Salt Water Taffy", 1.5, 209));
        checkout.enterItem(new Candy("Candy Corn", 3.0, 109));

        System.out.println("\nNumber of items: " + checkout.numberOfItems() +
"\n");
        System.out.println("\nTotal cost: " + checkout.totalCost() + "\n");
        System.out.println("\nTotal tax: " + checkout.totalTax() + "\n");
        System.out.println("\nCost + Tax: " + (checkout.totalCost() +
checkout.totalTax()) + "\n");
        System.out.println(checkout);
    }
}

```

## Output

Number of items: 4

Total cost: 1331

Total tax: 87

Cost + Tax: 1418

M & M Dessert Shop  
-----

2.25 lbs. @ 3.99 /lb.

Peanut Butter Fudge 8.98

Vanilla Ice Cream 1.05

Hot Fudge Sundae with	
Choc. Chip Ice Cream	1.95
4 @ 3.99 /dz.	
Oatmeal Raisin Cookies	1.33
Tax	.87
Total Cost	14.18

Number of items: 6

Total cost: 1192

Total tax: 77

Cost + Tax: 1269

M & M Dessert Shop

Strawberry Ice Cream	1.45
Caramel Sundae with	
Vanilla Ice Cream	1.55
1.33 lbs. @ .89 /lb.	
Gummy Worms	1.18
4 @ 3.99 /dz.	
Chocolate Chip Cookies	1.33
1.50 lbs. @ 2.09 /lb.	
Salt Water Taffy	3.14
3.00 lbs. @ 1.09 /lb.	
Candy Corn	3.27
Tax	.77
Total Cost	12.69

**Q4. (1 mark)** Given two matrices **A** and **B**, return the result of **AB**.  
You may assume that **A**'s column number is equal to **B**'s row number.

**Example:**

**Input:**

```
A = [
  [ 1, 0, 0],
  [-1, 0, 3]
]
```



```
B = [
  [ 7, 0, 0 ],
  [ 0, 0, 0 ],
  [ 0, 0, 1 ]
]
```

**Output:**

```

      | 1 0 0 |   | 7 0 0 |   | 7 0 0 |
AB = | -1 0 3 | x | 0 0 0 | = | -7 0 3 |
      | 0 0 1 |

```

**Q5. (Extra Credit- 2 mks)**

Given a string `S` that **only** contains "I" (increase) or "D" (decrease), let `N = S.length`.

Return **any** permutation `A` of `[0, 1, ..., N]` such that for all `i = 0, ..., N-1`:

- If `S[i] == "I"`, then `A[i] < A[i+1]`
- If `S[i] == "D"`, then `A[i] > A[i+1]`

**Example 1:**

**Input:** "IDID"

**Output:** [0,4,1,3,2]

**Example 2:**

**Input:** "III"

**Output:** [0,1,2,3]

**Example 3:**

**Input:** "DDI"

**Output:** [3,2,0,1]

**Note:**

1. `1 <= s.length <= 10000`
2. `s` only contains characters `"I"` or `"D"`.