CSG 100 Data Structures Fall 2004

Problem Set #2: Objects inside Objects inside Objects...

<u>Due Date: 21st of October</u>

Goal:

In this exercise you are to play the role of a developer who is given a specification and description of the data structures and their operations. You then have to provide a Java implementation for this specification.

Instructions:

For each exercise make sure you provide a `main` method to run your code as well as **all** the test cases that you have used to test your code. Comment all of your code and provide any information about your homework in a separate text file called README.txt. Send **all** your files to *skotthe@ccs.neu.edu*.

1) More fruits and baskets.

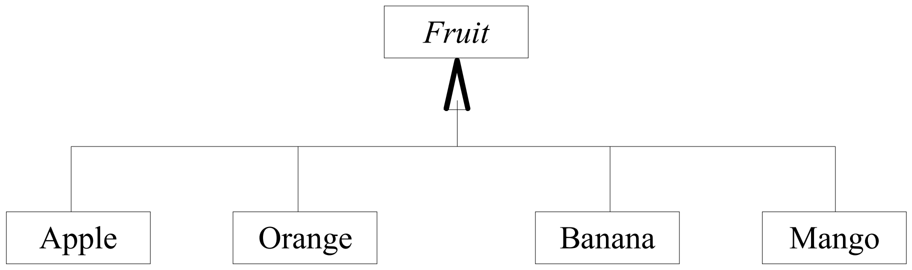


Figure 1: Class Hierarchy for Fruits

`Fruit` is an *abstract* class with two instance variables

- price of type `double`
- weight of type `double`

`Fruit` also provides the following methods along with their implementation:

```
1.getWeight():double
2.getPrice():double
3.setWeight(double):void
4.setPrice(double):void
```

The following method is defined as *abstract* in `Fruit` and all subclasses should provide their own implementation for it

```
1.prettyPrint():String
```

You are asked to implement all the subclasses of `Fruit` as shown in Figure 1.

You are now asked to implement different fruit basket offers

1. `FruitBasket`
    - can contain at most 10 fruits of any combination of types (e.g. `Apple`, `Banana`, `Mango` or `Orange`)
    - implements methods to add a fruit to the basket (i.e. `addFruit (Fruit):boolean`) and to remove a fruit from the basket (i.e. `removeFruit(int):boolean`). Both methods return `true` on success and `false` otherwise.
    - implements the method `showContents():String`, that prints out the contents of the `Basket`.
    - implements the method `getBasketWeight():double` that returns the total weight of the basket
    - implements the method `getBasketPrice():double` that returns the total price of the basket
1. `AppleBasket`
    - can contain as many apples as possible as long as the total price of the basket does not exceed $10.
    - the same methods as `FruitBasket` are also implemented by `AppleBasket` i.e. `showContents()`, `getBasketWeight()`, `getBasketPrice()`, `addFruit(Fruit)`, `remove(Fruit)`
2. `OrangeBasket`
    - can contain as many oranges as possible as long as the total weight of the basket does not exceed 10kg
    - the same methods as `FruitBasket` are also implemented by `OrangeBasket` i.e. `showContents()`, `getBasketWeight()`, `getBasketPrice()`, `addFruit(Fruit)`, `remove(Fruit)`

(Hint: Take advantage of the repetitive behavior that all baskets must have)

Finally implement `SaleTrolley` which can take at most 10 baskets. `SaleTrolley` has to also provide the following methods

1. `showTrolleyPrice():double`, providing the total cost of all the fruits found in the trolley.
2. `showTrolleyWeight():double`, providing the total weight of all the fruits found in the trolley.
3. `showTrolleyContents():String`, that returns as a string a pretty printing of all the baskets with their contents.

(30 Points)

2) Manipulating shapes.

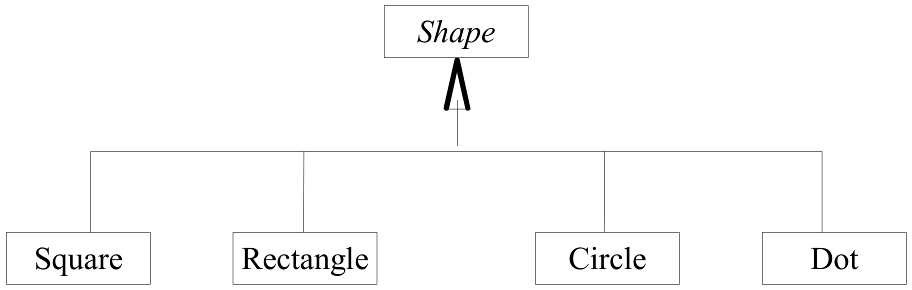
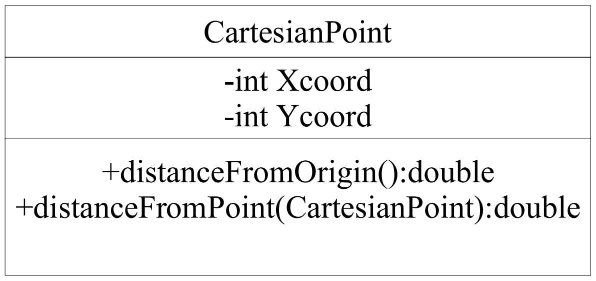


Figure 2: Class Hierarchy of `Shapes`.

The class `Shape` is an abstract class with one instance variable of type `CartesianPoint` (Figure 3). `Shape` defines the following public *abstract* methods:

1. `area():double`, returns the area of the `Shape`.
2. `distanceToOrigin():double`, the distance from (0,0) to the Shape's `CartesianPoint`
3. `distanceToShape(Shape):double`, distance from this Shape's `CartesianPoint` to the argument's `CartesianPoint`
4. `in(Dot):boolean`, return `true` if the `Dot` given as argument is within the area of the Shape.
5. `draw():String`, this method return the information of a `Shape` in the form of a string.

`CartesianPoint` is defined as:



CartesianPoint
-int Xcoord
-int Ycoord
+distanceFromOrigin():double
+distanceFromPoint(CartesianPoint):double

Figure 3. Class Diagram for CartesianPoint.

You are asked to provide the implementation for each of the subclasses of `Shape`. You should provide the implementation for each of the inherited abstract methods but also add any new instance variables and methods that you need in order to calculate the area of each shape. You might find `java.lang.Math` helpful.

(30 Points)

3) Implement a ToDo List. A ToDo list can have many levels of nesting. The first level of nesting consists of `ToDoEntries` which hold the date of entry, the due date for the task as well as a list of sub-items that need to be done for this task. Sub-items can be any valid string that holds information about what to do. For example my ToDo List is

Added: 10/15/2004

Due : 17/15/2004

Id: CSG100

Items:

1) Check final class list for csg100
2) Check homework2
3) Update the Web page

   Items:

   1. New Slide
   2. New Homework
   3. New Links for IDEs

Added: 10/14/2004

Due: 10/16/2004

Id: Conferences

Items:

1) Check Conference dates
2) Submit proposal for demo
3) email details

Added: 09/30/2004

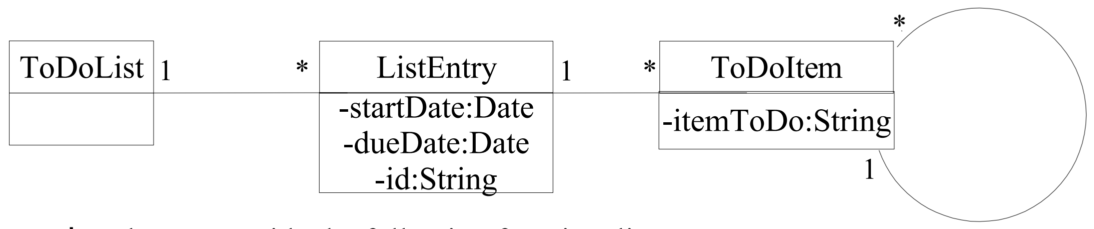
Due: 12/20/2004

Id: Personal

Items:

1) check plane tickets

The following Class Diagram is to help you get started.



`ToDoList` has to provide the following functionality:

1. `addListEntry(ListEntry):boolean`, adds a new `ListEntry` to the ToDoList
2. `addToDoItem(Id, ToDoItem):boolean`, adds `ToDoItem` to the `ListEntry` with `Id`.
3. `showList():String`, pretty print the whole list and return the result as a `String`
4. `showOnlyToday(Date):String`, pretty print the part of the list that is due today as a `String`
5. `showOnlyTomorrow(Date):String`, pretty print the part of the list that is due tomorrow and return it as a `String`

Define `Date` as a Java class that holds three integers, *month*, *day* and *year*. Define also a pretty-print method on `Date`.

(40 Points)