SENG2200 Programming Languages & Paradigms
School of Electrical Engineering and Computing
Semester 1, 2019

Assignment 2 (100 marks, 15%) – Due Friday May 3, 23:59

1. Objectives

This assignment aims to build the understanding of Java programming in topics of *interface, polymorphism, generics* and *iterator*. A successful completion should be able to demonstrate a solution of assignment tasks with correct Java implementation and report.

2. Required Solution

The firm in Denman still believes that data structures that are hand-coded will run faster than those used from the Java libraries and so this is still required for your container types. However, once the true nature of generic structures was explained to them, they have decided that containers will use generic specifications and standard iterator *interfaces*.

2.1 The `Point` class should be exactly the same as for assignment 1

The **`Point`** class simply has two floating point values for x and y coordinate values. It should have a method that will calculate the distance of the point from the origin. Your **`Point`** class should also contain a **`toString()`** method which will allow the conversion of a **`Point`** object into a String of the form **(x_i, y_i)** – include the open and close parentheses in the String and use the same **`4.2f`** format as for **PA1**. This will be used for output of your results.

2.2 Designing with a view to extending your program in the future

We know from assignment 1 that we will need to deal with polygons, and we know how to do that. However, it doesn't take much thought to realize that there are many shapes other than polygons (*even when we remember that rectangles and squares are also polygons*). So a completely different approach is needed to the design. This will include an **abstract class** that other classes will **inherit** from.

Write an *abstract* **`PlanarShape`** class. It will have an *abstract* **`toString()`** method for printing results, an *abstract* **`area()`** method and an *abstract* **`originDistance()`** method.

Note that even though we know how to implement each of these for **`Polygon`** which have been modelled as a sequence of vertices, we cannot implement them for other **`PlanarShapes`** until we know exactly what these shapes are, and how they are to be represented in the program.

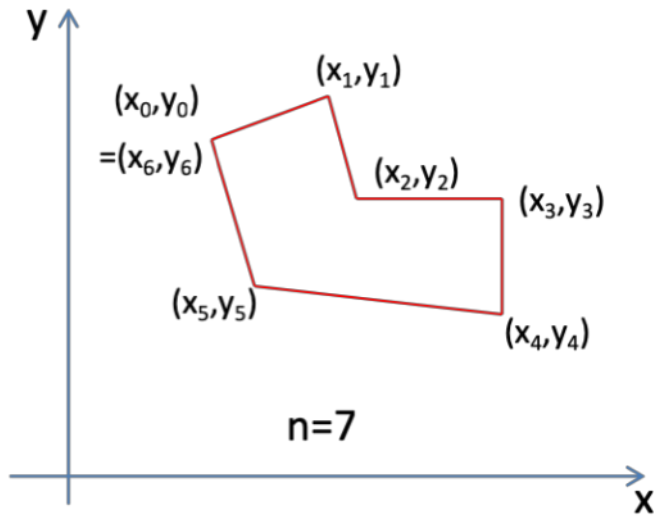
`PlanarShape` has an ordering based on **`area()`** and **`originDistance()`** as previously was the case for **`Polygon`**. If any two **`PlanarShape`** objects have areas *within 0.005 units* of each other, then they are assumed to have *equal area*, in which case, the planar shape with the lower **`originDistance()`** takes precedence.

Because we need to be able to *compare* **`PlanarShape`**, we also need to use the standard **`Comparable<T>`** *interface*, using the *specification* **`implements Comparable<PlanarShape>`**.

The **`Polygon`** class is similar to Assignment 1 but will require a re-design because of the above. As per **PA1, t**he area of a polygon, specified on the Cartesian plane by its vertices, is given as follows:

$$A = \frac{1}{2} \left| \sum_{i=0}^{n-2} (x_{i+1} + x_i)(y_{i+1} - y_i) \right|$$

e.g.



Note that for a six sided figure, such as the one shown, *n* is *7*, because the last point describing the polygon is equal to the first (it is always the same Cartesian point).

The special thing about a **`Polygon`** now, is that it is a **`PlanarShape`**. The **`Polygon`** class has an *array* of **`Point`** objects representing the vertices of the polygon.

Your **`Polygon`** class should contain a **`toString()`** method which will allow the conversion of a **`Polygon`** object into a *String* of the form **POLY=[point$_0$…. point$_{n-1}$]: area_value** and this will use the same format for area as **PA1**. This will be used for output of your results.

The **`Polygon`** class also has an implementation of the **`area()`** method, given by the formula above. We will now have to implement the method **`originDistance()`** for this class as well, calculated as it was in **PA1**. The polygons have an ordering based on area and **`originDistance()`** as before, however this will be done by way of the **`PlanarShape`** class.

When data for a polygon is input it will be of the same form, e.g.

```
P  6  4  0  4  8  7  8  7  3  9  0  7  1
```

As was used in the first assignment, that is, the letter P, then the number of sides (6), then 6 pairs of values for the 6 vertices.

2.3 Re-design the Linked List classes using generics and iterator(s)

You are also required to implement a *circular doubly-linked list*, using a *single sentinel node* to mark the start/finish of the list, which you should call **`LinkedList`**. It should contain methods to **`prepend`** and **`append`** items into the list and a means of taking items from the **`head`** of the list. It will *not need a current* item because any functionality requiring this should migrate to the **`iterator`** class for the **`LinkedList`**.

The linked list previously just used **`Object`** references and casting to insert or access polygon objects into the list (or **`Polygon`** references to be explicitly stored in the list). We now require **`PlanarShape`** objects (references to them at least) to be the basis of the list, and we need to *add type protection* for this by way of a generic specification of the list and instantiation of the list so that it *only allows* **`<PlanarShape>`** objects to be inserted and accessed. Your **`LinkedList`** must be **`Iterable`** and so you need to provide an iterator for your **`LinkedList`** class. *This will only implement the standard iterator methods*.

The second list in your program will now be a **`SortedList`**, which can be designed as *an extension of the first list*. It will need an **`insertInOrder()`** method (and any supporting attribute data) that will allow construction of the list by means of the ***insertion sort algorithm***. It will need to properly use the **`comparable`** interface implementation of the **`compareTo<PlanarShape>`** method from within the **`PlanarShape`** class. Your **`SortedList`** will also be instantiated to only hold **`PlanarShape`** objects.

3. Program PA2a

Program **PA2a** simply uses the same data as was used for **PA1**, and produces similar output (except for the slight **POLY=** change in the **toString()** output generated).

The first main task is to read polygon specifications (until end of file, from standard input) and place them into an instance of your circular list, in input order.

Each **Polygon** will be specified in the input by the letter **P**, followed by the number of sides the polygon has (ie n-1), and then pairs of numbers which represent the respective vertices on the Cartesian plane (*x-value then y-value*), which means vertices p_0 to p_{n-2} from the above formula. **You do not have to worry about any of the data being missing, or out of order.** It will probably be best for your polygon objects to contain all n points, that is, explicitly including the last vertex as a copy of the first.

You are then to produce a second list, which contains the **Polygon** objects, sorted into *"increasing area/origin-distance order"*. This is to be done by iterating through the first list and placing the polygons into the second list using an *insertion sort algorithm*.

However, notice that at no time does your program need to know that these are **Polygon** objects – the design simply refers to them as **PlanarShape** objects, and each **PlanarShape** decides that it is a **Polygon** and responds to any request accordingly. The **PlanarShape compareTo()** method will be able to fetch the **Polgon** values of **area()** and **originDistance()** because of the *polymorphism* capability that Java provides.

3.1 Output of PA2a

Output for **PA2a** is a complete print of both your lists (i.e. the polygons in input order, and then the polygons in sorted order, listing the area of each. The output should also be produced using *iterators* to visit the objects in the lists.

3.2 Factory Pattern

Another key to making this design easy to extend is for **PA2a** to use a factory method to produce polygon objects and return them using a **PlanarShape** reference. For example

```
PlanarShape shapeFactory( )
{
     PlanarShape shape;
     Read the identification character;
     switch (id_char)
     {
     case 'P':
          read in polygon data;
          shape = new Polygon (...data as input...);
          return shape;
```

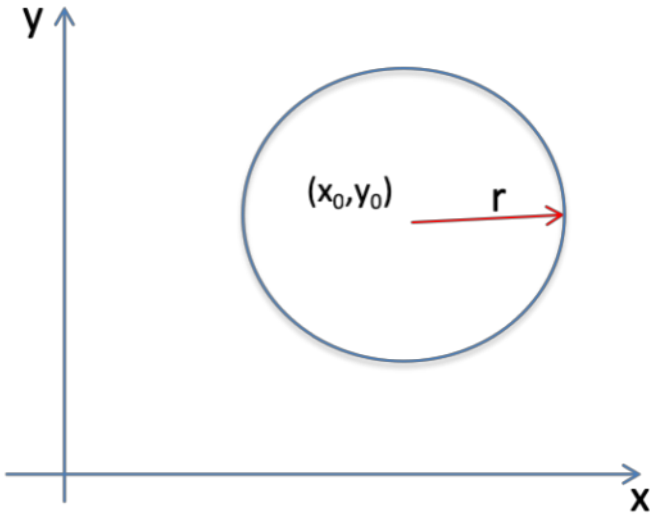
```
            break;
        default:
            Report an error – invalid input type;
        }
}
```

4. Program PA2b

The second program should now be able to make use of this design to *extend* the program so that it can process both *circle objects* and *semi-circle objects*, while making minimal changes to code written so far. Almost all the new code for these objects will reside in separate new classes.

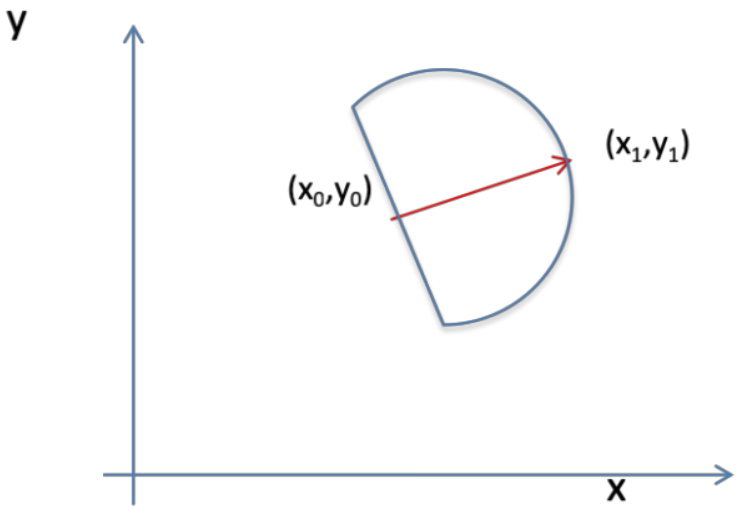
4.1 Circle

`Circle` objects are specified by a **`Point`** which is the centre of the circle and a floating point value which is the radius of the circle. The area of a circle is the usual pi-r-squared calculation, while its origin distance is simply the distance of the centre from the origin. Its **`toString()`** method produces the string **CIRC=[point$_0$ radius]: area_value**. Input data for a circle object is C x0 y0 r. The **`originDistance()`** of a circle is given as the distance from the origin of the centre minus the radius (*possibly negative*).



4.2 Semi-Circle

`SemiCircle` objects are specified by a **`Point`** which is the centre of the base of the semicircle, and a second **`Point`** which specifies the point on the semi-circle at the end of the perpendicular to the base.



The radius of the semi-circle is the length of the perpendicular vector, and so the area of the semi-circle is simply $\pi r^2/2$. It's **`toString()`** method produces the string **SEMI=[point$_0$ point$_1$]: area_value**. An input data specification for a semi-circle object is S x_0 y_0 x_1 y_1. The **`originDistance()`** of a semicircle is given as the distance from the origin of the closest of the two data points and the two base extremity points.

By extending your factory method of program **`PA2a`** to construct the objects required, no further changes should be needed to the classes from **`PA2a`** to **`PA2b`**.

5. Written Report

As you design, write and test your solution, you are to keep track of and report on the following:

1. For each of the programs keep track of how much time you spend designing, coding and correcting errors, and how many errors you need to correct.
2. Keep a log of what proportion of your errors come from design errors and what proportion from coding/implementation errors.
3. Provide a (*brief*) design of how you would further extend your **PA2b** so that it specifically included **`Triangle`** and **`Square`** figures, with their own '**`T`**' or '**`Q`**' input designations respectively. Draw the UML class diagram for this new program (*intricate detail not required*). What attribute data do you need in each case?
4. Investigate the mathematical structure of an **`Ellipse`** on the Cartesian plane. How would you model the **`Ellipse`**? How would you then calculate its area and **`originDistance()`**? How would this be incorporated into your program? Draw another UML class diagram to show this.

6. Submission

Submission is through the Assessment tab for SENG2200 on Blackboard under the entry Submit PA2. If you submit more than once then only the latest will be graded. Every submission should be ONE ZIP file named c9999999PA2.zip (where 9…9 is your student number) containing:

- Assessment item cover sheet.
- Report (PDF file): addresses the tasks of Section 5 (around 2-3 A4 pages).
- Program source files (e.g., driver classes **PA2a** and **PA2b**).

6.1 Coding Language and Compilation

Programming is in **Java**, and will be compiled against **Java 1.8**, as per the standard lab environment.

Name your startup class **`PA2a`** and **`PA2b`** (capital-P capital-A number-2), that is, the marker will expect to be able compile your program with the command:

`javac PA2a.java` and **`javac PA2b.java`**,

…and to run your program with the command:

`java PA2a test.dat` and **`java PA2b test.dat`**,

…within a **Command Prompt window** (where **`test.dat`** could be any valid filename).

6.2 Notes

Your application will run and take data from a standard text file (*in the form given above)* which will include multiple shape definitions – you may assume that the number of points specified for the shape will always be correct – and will be specified from the command line, in form **`java PA2a test.dat`**
It is expected that **`PA2a`** and **`PA2b`** will

- be in the *root of your submission folder*
- *compile the entire project* without any special requirements (including additional software).

You may include a **`readme.txt`** file, if you require any special switches or compilation method to be used; but this may incur a penalty if you stray too far from the above guide lines.

You may only use Java libraries for input and output. All input to the program will come from *standard input*. All output is to *standard output*.

7. Warning

You will be tempted to copy code already implemented for Assignment 1. This may cause significant problems as your program design has not followed the best Object-

Oriented analysis techniques, and this is likely to present problems as you update code. Always be ready to go back to square 1 and start again on your implementation.

8. Marking Criteria

*** If your submission cannot be compiled or has run-time errors (e.g., segmentation fault), you may receive ZERO mark.*

*** The mark for an assessment item submitted after the designated time on the due date, without an approved extension of time, will be reduced by 10% of the possible maximum mark for that assessment item for each day or part day that the assessment item is late. Note: this applies equally to week and weekend days.*

This guide is to provide an overview of Assignment 2 marking criteria. It briefly shows marks of each section. This guide is subject to be adjusted.

Program Correctness (80%)

- ***Comparable*** Interface **(5%)**
- ***Polymorphism*** *implementation* **(20%)**
  - *Incl. required methods*
  - *Incl. factory pattern (method)*
  - *If polymorphism concept has not been correctly implemented, you will significantly lose mark here.*
- ***Generics*** **(10%)**
  - *Your program should properly use "generics" concept in (at least) the required classes*
- ***Iterator*** *implementation* **(20%)**
  - *Implementation of standard Iterator interface*
  - *Each required method implementation may have different marks.*
- ***Input/Output*** **(10%)**
  - *Your program should be able to correctly read text (shape) specifications.*
  - *There could be differences in your overall output format, however, any format specified, e.g., 4.2f, MUST be correctly implemented.*
- ***General*** **(15%)**
  - *Incl. required general methods, such as area calculation and sorting.*
  - *OO design – your program should follow the **basic** principles of OO design, while you will NOT be marked specifically on the design.*

Report (10%)

Miscellaneous (10%)

- Code Format & Comments **(10%)**
  Note: the issues from either code format or comments may cause the maximum deduction from this component.