The University of Newcastle
School of Electrical Engineering and Computer Science
SENG2200 Programming Languages & Paradigms
Semester 1, 2016
Programming Assignment 2 - Due May 1, 23:59pm

Submission is to the ***Assignments Tab*** for SENG2200 on Blackboard. This allows you to submit the assignment for it to be available for grading. If you find errors after submission, then you may re-submit your assignment, but only the last submission will be graded. The written report is to be submitted as part of the online submission outlined below, and so you need to finish the program(s) with enough time to spare (I suggest April 28) in order to complete the written report. Remember to include an Assessment Item Cover Sheet with your submission.

You are to write two programs in Java, these programs will have driver classes called **PA2a** and **PA2b**. You are to submit a **.zip** file called **c9999999a2.zip** to the assignments tab and this file should contain all the source code to allow the marker to compile and execute your program. The file should contain an assignment cover sheet and the written report.

For this assignment, the written report will be worth 10% of the assignment mark. It should still be producible in about 2 or 3 A4 pages of work.

A 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.

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.

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 $\mathbf{(x_i , y_i)}$ – 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.

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

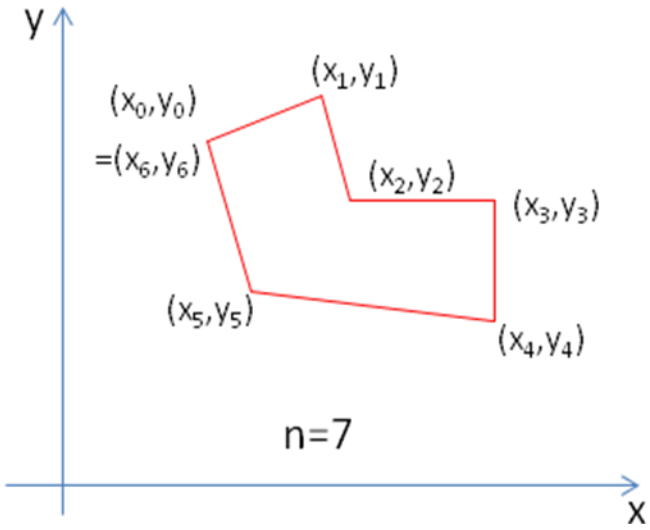
polygons). So a completely different approach is needed to the design. This will include an abstract class that other classes may inherit from.

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 polygons which have been modelled as a sequence of vertices, we cannot implement them for other Planar Shapes until we know exactly what these shapes are, and how they are to be represented in the program. The Planar Shapes have an ordering based on ***area( )*** and ***originDistance( )*** as previously was the case for polygons. If any two ***PlanarShape*** objects have areas within 0.05% of each other, then they are assumed to have equal area, in which case, the planar shape with the lower ***originDistance( )*** takes precedence (comes first in the ordering). Because we need to be able to compare planar shapes, 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 requires re-design because of the above: The 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.

The Linked List class(es) will also require re-design to use 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 a sorted list by means of the insertion sort algorithm. Your SortedList will also be instantiated to only hold ***PlanarShape*** objects. It will need to properly use the comparable interface implementation of the ***compareTo***<***PlanarShape***> method from within the ***PlanarShape*** class.

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 most simply 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 the objects as planar shapes and each planar shape 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.

Output for PA2a is a complete print of both your lists (ie 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.

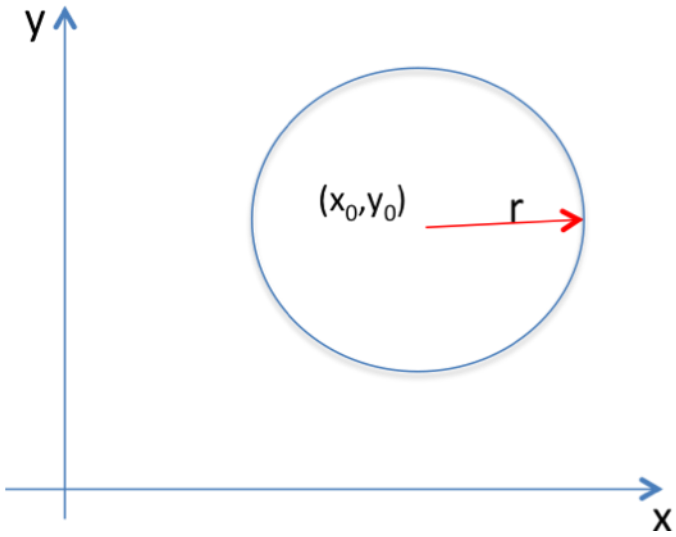
Another key to making this design easy to extend is for PA2a to use a factory method to produce polygon objects and return them using a ***PlanarShape*** reference. For example

```
PlanarShape shapeFactory( ) {
     PlanarShape shape_ref;
     Read in the leading (identification) character;
     switch (id_char) {
     case 'P':
          read in polygon data;
          shape_ref = new Polygon (……data as input ….);
          return shape_ref;
          break;
     default:
          Report an error – invalid input type;
     }
}
```

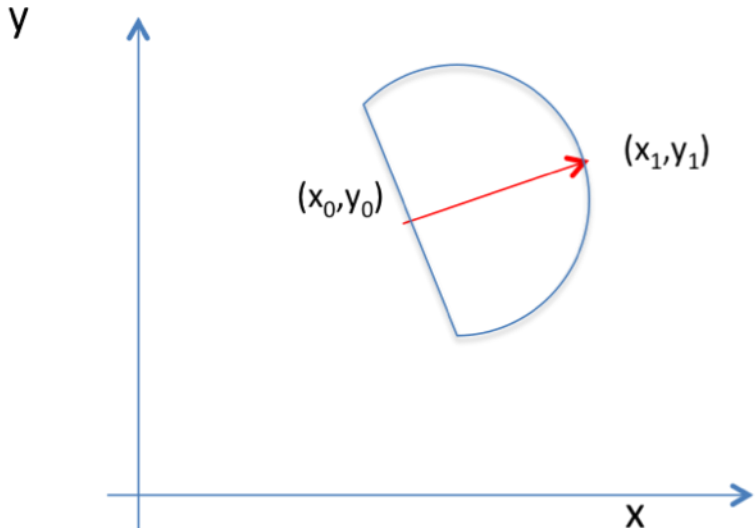
Program PA2b:

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.

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). *(/over)*



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-squared-over-2. Its ***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. *(/over)*

The 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.

M.R. Hannaford 21 March 2016.