Lab 3

Objectives

- Introduction to interfaces
- Introduction to abstract data types

Part I - Interfaces

1. Download all of the provided lab files to your Lab3 working directory.

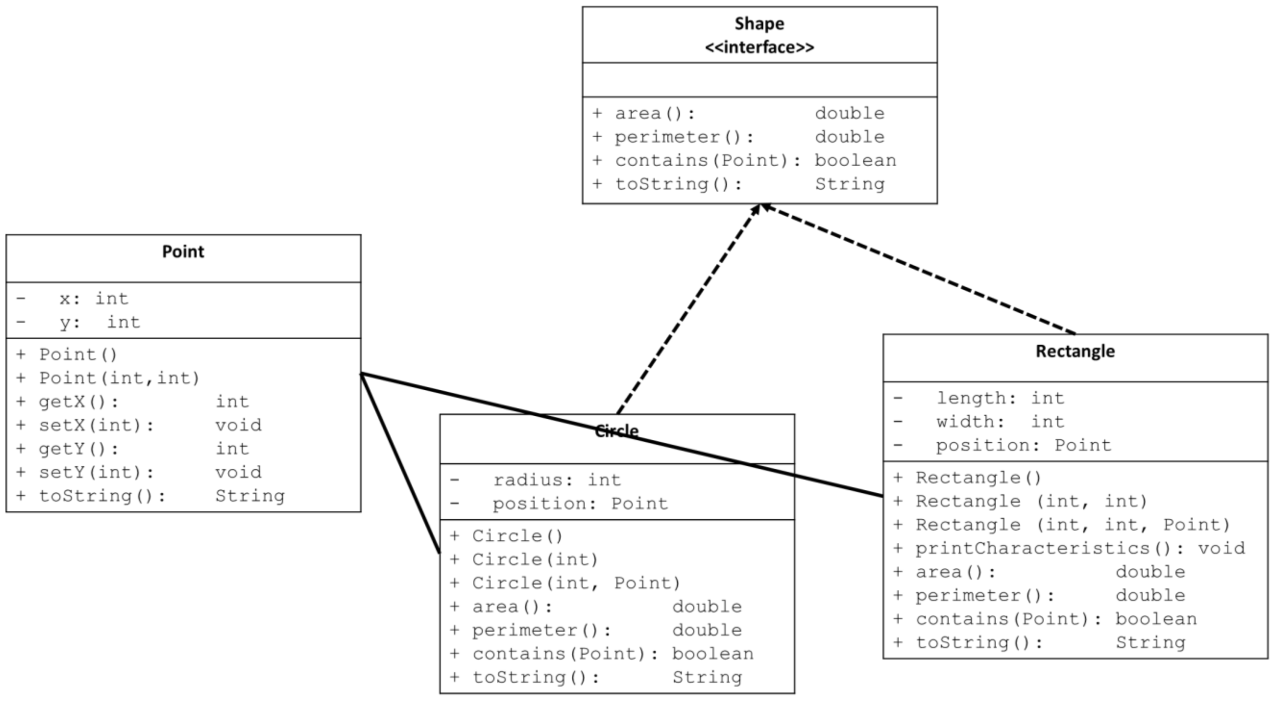
We are going to use an interface to specify what a Shape class should look like. As depicted in the UML diagram below, a Shape should have methods for calculating area, calculating perimeter and determining whether a given Point lies within a Shape. An interface does not include the implementation of these methods, only their signatures and documentation. The implementation details of these methods is dependent on the type of Shape and therefore will be defined in the classes that implement the Shape interface (ie. Circle, Rectangle)

2. Start by opening Shape.java to familiarize yourself with the method documentation.

The Circle and Rectangle classes *implement* the Shape interface (depicted by the **dashed arrow** between them in the UML diagram below). Notice, the Rectangle class has the method printCharacteristics which Circle does not **but** both Circle and Rectangle **must** have implementations of all methods specified in the Shape interface (area, perimeter, contains and toString).

Circle and Rectangle both have an attribute of type Point (depicted by the **solid line with no arrow** between them in the UML diagram below).

3. Open Circle.java and you will see a full implementation of the Circle class. Notice, it contains implementations of all methods listed in Shape.java



4. Begin the implementation of Rectangle.java
    a. Open Rectangle.java
    b. Try to compile Rectangle.java – You will see compile error something like this:

    ```
    error: Rectangle is not abstract and does not override abstract method contains(Point) in
    Shape
    ```

    Fix this compile error by introducing stubs for each of the methods Rectangle must implement:

    c. Copy the documentation and method signatures from Shape.java to Rectangle.java
    d. For each method signature:
        i. remove the "**;**" from the end of the signature
        ii. add an "**{**" and "**}**" to make it a method
        iii. make the method public
        iv. if the method that has a non-void return type, add a dummy return statement

    For example, for the area method the stub would look like this:

    ```
    public double area() {
            return 0;
    }
    ```

    NOTE: we gave you the implementation of the constructors and the toString method in the Rectangle class – do not change them.

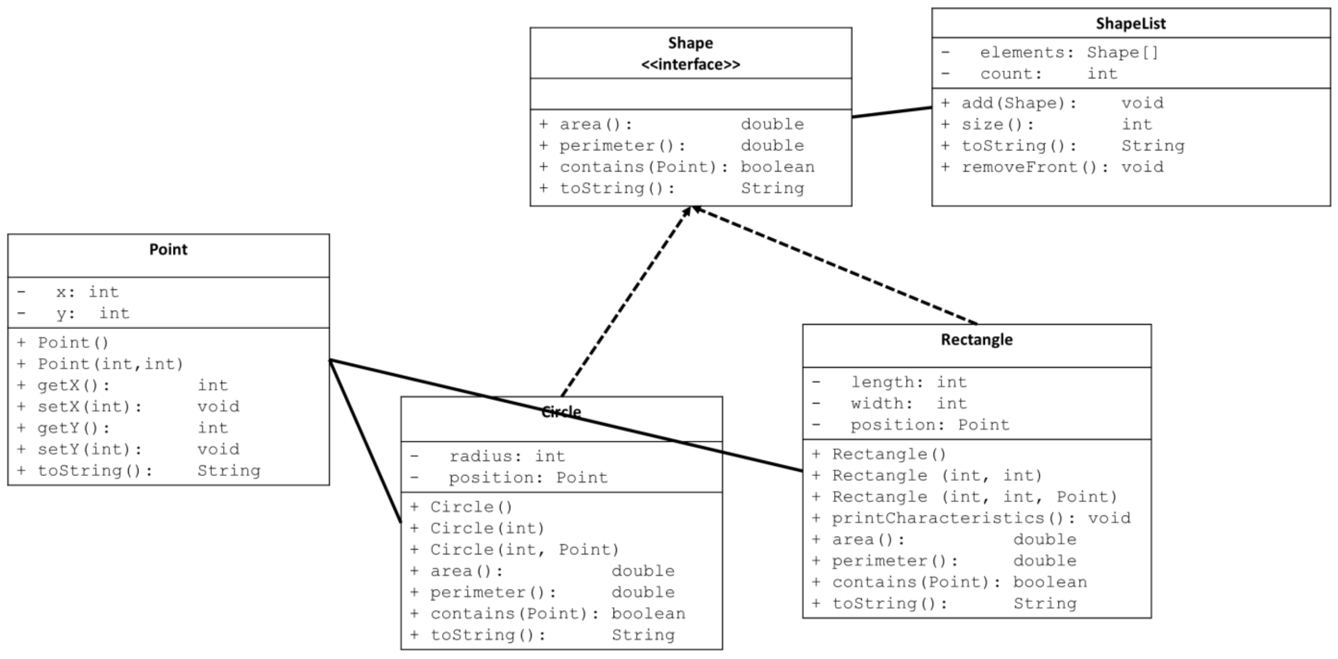
CHECKPOINT 1 – Now might be a good time to check-in with the TA if you are unclear of how to proceed, or if you are unable to fix the compile errors in your program.

5. Complete the implementation of Rectangle.java
    a. Open Lab3Tester.java and compile and run it. You will see tests failing.
    b. Implement one method at a time, recompiling and rerunning the tester after each one

CHECKPOINT 2 – Get help from the TA if you are unable to complete this part.

Part II – Using classes to define a data structure

We are now going to implement a class called ShapeList that will hold a collection of Shapes. The addition of this class is depicted in our UML diagram:



1. Start by opening ShapeList.java Notice we have added the attributes, a blank constructor and method stubs for you
    a. Uncomment the call to testShapeList() in the main of Lab3Tester.java and compile/run it
    b. The ShapeList constructor has been implemented for you. Notice it initializes the two attributes of the shape list. You will use those attributes to implement the other methods.
2. Read through the provided tests in testShapeList() to ensure you understand the function of each of the methods.
    a. Uncomment the print statements to help you when debugging your implementation.
    b. It is up to you to decide how to handle the case when the underlying array is full. A new larger array should be allocated, and all elements should be copied into the new array.

CHECKPOINT 3 – Main Lab Complete (extra practice activities can be found on the following page)

Additional Practice – Implementing the removeFront method

1. Make sure to implement the removeFront method so that it has the correct behaviour:
   a. The first element should be removed, the size should go down one, and the order of all other elements should not change. There should be no gap at index 0.
   b. If there are no elements to remove, nothing should happen (the size stays at 0)

2. **BONUS CHALLENGE:** If the array is less than half full, allocate a smaller array and copy the elements over to it, so there is less "wasted" memory.