Department of EECS
York University

Term: Fall 2017
Instructor: Andy Mirzaian

EECS2011: Fundamentals of Data Structures
Section A
Assignment 1
Due: 10 pm, Friday, September 29, 2017

Instructions:

- This is an individual assignment; you may not share code with other students.
- Read the course FAQ on the generic marking scheme and how to submit assignments electronically. The submission deadline is strictly enforced.
- This assignment consists of 3 problems and is designed to let you practice on arrays, (nested) loops, and some elementary constructs of the object oriented Java programming language.
- Here you can access the directory of this assignment, **A1** (which is also the package name). It includes the file **a1.pdf** that you are now reading, and some code templates that you may use to complete your assignment solutions.
- Print your name, eecs account, and student ID number on top of **every** file you submit. Space is provided at the top of each code template to enter your identification. We are not responsible for submitted files that have no student identifications.
- You may add more classes/fields/methods and test cases of your own as required.
- Use informative Javadoc-style comments as well as inserted comments to further explain your code.
- In addition to the required .java files, you should also submit a file named **a1sol.pdf** with the following contents:
  - Provides further explanations and illustrations (particularly pertaining subtle parts of your codes) to help the reader better understand the underlying ideas behind your codes.
  - Includes input/output results of your programs (by copy-and-paste from the console) for each of the 3 problems.
  - Explains, to the best of your ability and knowledge, why your solutions satisfy the stated specifications such as correctness, running time and memory space efficiency.

Problem 1. [32%] Finding a duplicate in a preconditioned array

Implement the *findDuplicate()* method in the *ArrayDuplicateElement* class so that it performs as indicated in the commented specifications. The *main()* method in this class runs some test cases on your method. You should also add a few nontrivial and interesting test cases of your own at the end of *main().*

Further explanation:

findDuplicate() takes an integer input array *ints[0..n-1]* with n elements with the **pre-condition** that each array element is an integer in the range *1..n*. Either all n elements of *ints[ ]* are distinct, i.e., all integer values *1..n* appear in *ints[ ],* or *ints[ ]* contains one or more duplicate element values. This method arbitrarily selects and returns one such duplicate value if there is any; otherwise, it returns the special value -1 indicating that all elements of *ints[ ]* are distinct. Your method is allowed to change the input array, but the output should be with respect to the originally given input. (If the client does not wish their array to be altered by this method, they can pass a clone of it and keep their original array intact.)

Solution Requirements:

There are many obvious solutions to this problem that are inefficient in either running time or memory space utilization. For instance, the method may take time that is quadratic in input size, or may require additional large working storage such as arbitrary size arrays or other elaborate structures.

For full credit, your solution should take only **linear time**, i.e., *O(n)* time, and be **in-place**, i.e., *O(1)* space.

The word *"in-place"* means: in addition to the input, the method is allowed to use, as its work space, only a constant number of primitive variables or pointers to existing objects. This precludes the use of any additional non-primitive objects that are capable of holding more than a constant number of memory cells, such as arbitrarily large arrays. For example, if the input is an array, we may choose to use a few index variables, etc. to work "in-place" on this array.

File to be submitted: *ArrayDuplicateElement.java*

Problem 2. [36%] Longest Almost Flat Subarray

Implement the *longestAFS()* method in the *LongestAlmostFlatSubarray* class so that it performs as indicated in the commented specifications. The *main()* method in this class runs some test cases on *your method*. You should also add a few nontrivial and interesting test cases of your own at the end of *main().*

Further explanation:

The input is an **arbitrary** integer array ints[ ]. A contiguous subarray of ints is called **almost flat** if no two elements of that subarray differ by more than one. The method *longestAFS()* takes any integer input array *ints[0..n-1]* with n elements and returns a description of its longest almost flat subarray in the form of a two-element array *{start, len},* where *start* is the starting index of the subarray and *len* is its length. The method **should not** change the input array. If there are several such optimum subarrays, all with the same maximum length, break the tie by selecting the one with minimum starting index.

Solution Requirements:

There are many obvious solutions to this problem that are inefficient in either running time or memory space utilization. For instance, the method may take time that is quadratic or worse, or may require additional large working storage such as arbitrary size arrays or other elaborate structures.

For full credit, your solution should take only **linear time**, i.e., *O(n)* time, and be **in-place**, i.e., *O(1)* space.

The word *"in-place"* is explained in Problem 1.

File to be submitted: *LongestAlmostFlatSubarray.java*

Problem 3 . [32%] A Hierarchy of Planar Shapes

Design the hierarchy of embedded planar shape types as shown and illustrated in the figures on the next two pages.

An axis-parallel ellipse with horizontal axis a, vertical axis b, and center point with coordinates (x_c, y_c) is given by the formula:

$$\left(\frac{x - x_c}{a}\right)^2 + \left(\frac{y - y_c}{b}\right)^2 = 1 .$$

Notes:

1) The Java Library already has abstract class *Point2D,* and its nested class *Point2D.Double*. You are allowed to use them.
2) The Java Library also has classes *Shape* and *Ellipse2D.* You are **not** allowed to use them in any way for the solution to this assignment problem.

Further explanation:

The Ellipse and Circle constructors are overloaded: one as zero-parameter version and another as full-parameter version.

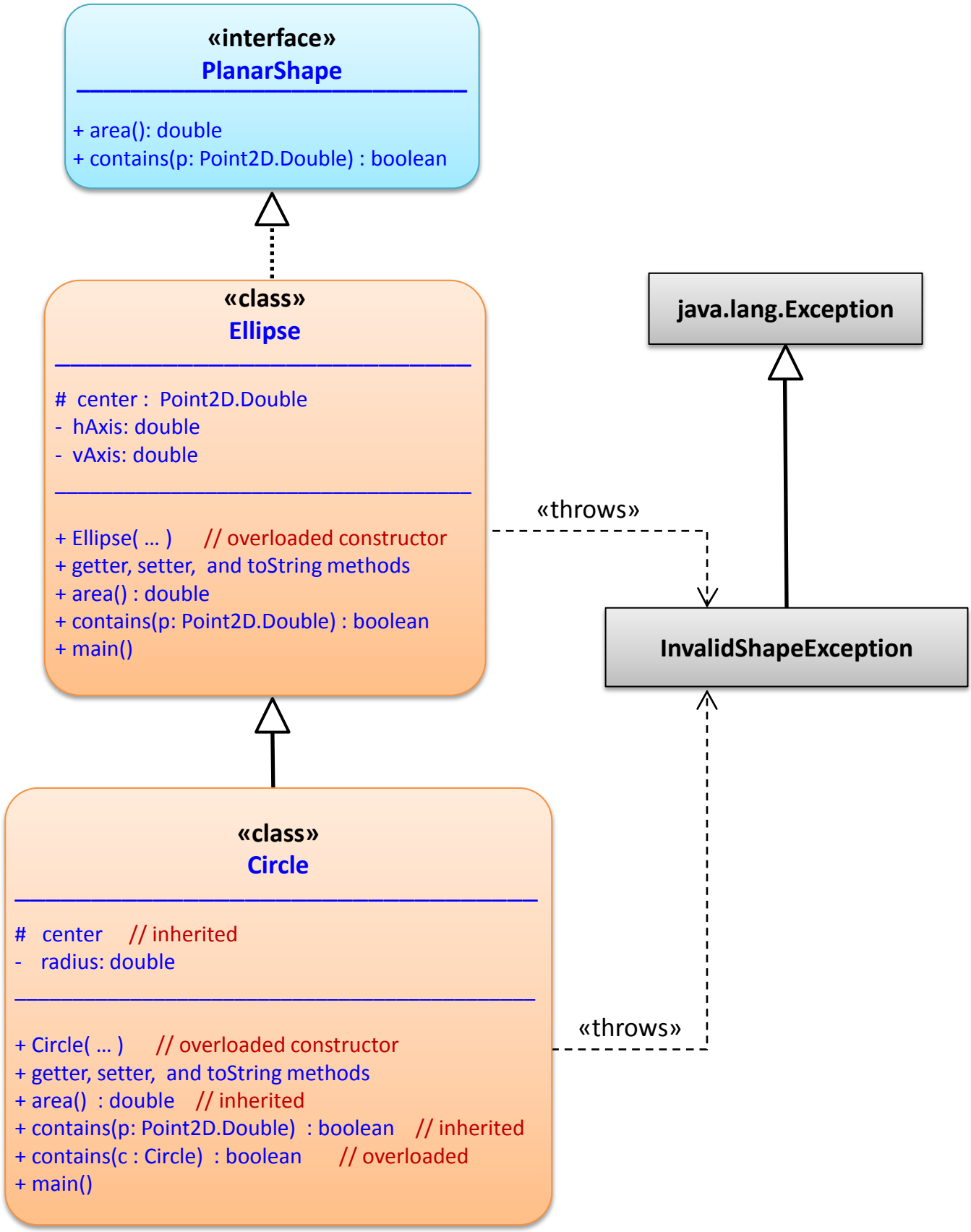
Circle inherits and possibly overrides the methods *area()* and *contains(Point2D.Double)* from its super-class Ellipse. Circle also has the overloaded method *contains(Circle).* The *contains()* methods return true if and only if the instance shape contains the parameter object entirely within its interior or on its boundary.

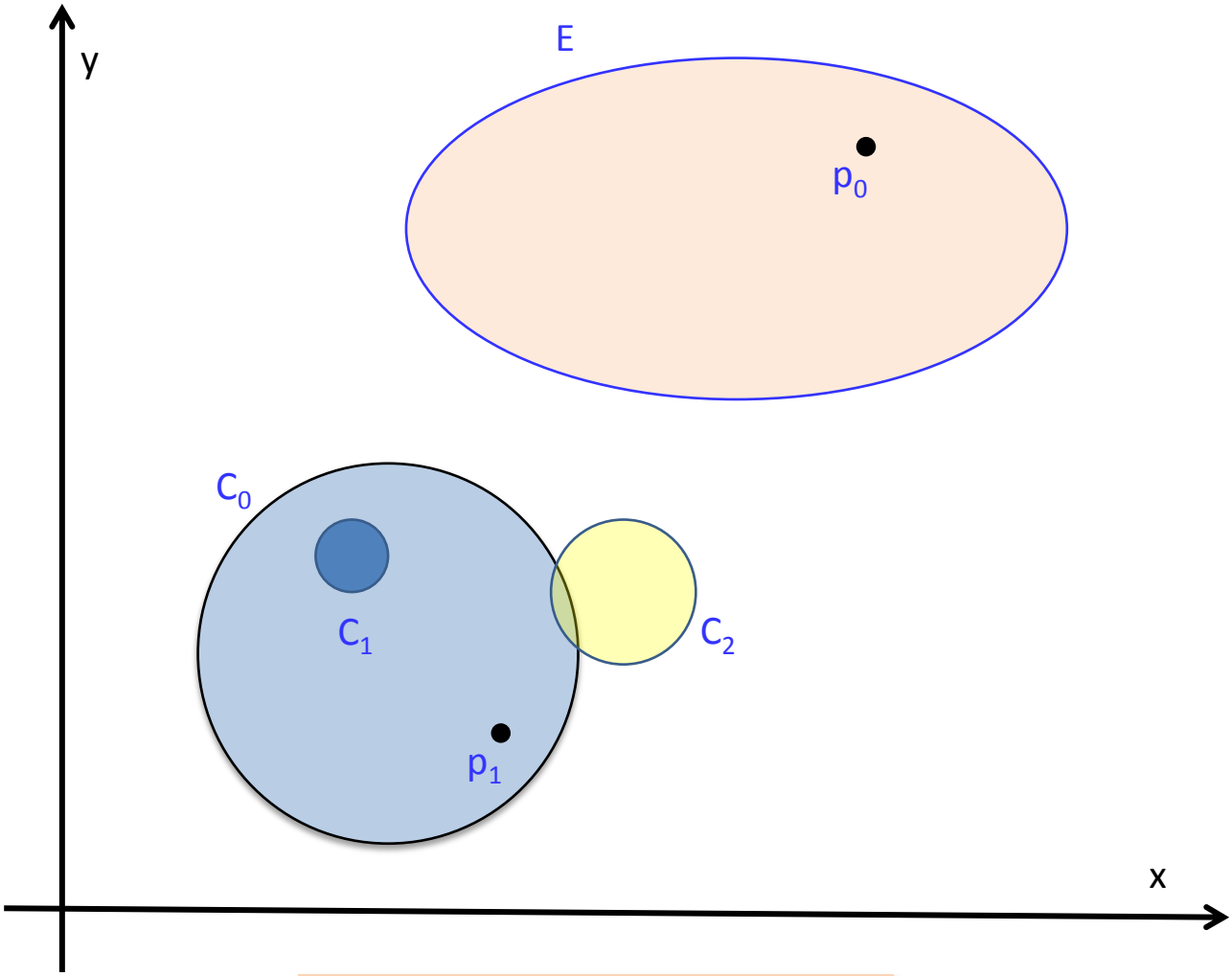
You should also design *InvalidShapeException* which is thrown by the constructor and the setters with an informative message when a shape with illegal parameters (e.g., negative ellipse axis or negative circle radius) is attempted.

The main() methods of Ellipse and Circle should thoroughly test these classes, including handling of InvalidShapeException that are thrown by the various methods. These tests should produce informative I/O that you should record in your a1sol.pdf file.

Files to be submitted:

- *PlanarShape.java*
- *Ellipse.java*
- *Circle.java*
- *InvalidShapeException.java*


«interface»
PlanarShape
+ area(): double
+ contains(p: Point2D.Double) : boolean
«class»
Ellipse
center : Point2D.Double
- hAxis: double
- vAxis: double
+ Ellipse(…) // overloaded constructor
+ getter, setter, and toString methods
+ area() : double
+ contains(p: Point2D.Double) : boolean
+ main()
java.lang.Exception
«throws»
InvalidShapeException
«class»
Circle
center // inherited
- radius: double
+ Circle(…) // overloaded constructor
+ getter, setter, and toString methods
+ area() : double // inherited
+ contains(p: Point2D.Double) : boolean // inherited
+ contains(c : Circle) : boolean // overloaded
+ main()
«throws»



Ellipse E contains point p_0 but not p_1.
Circle C_0 contains point p_1 but not p_0.
Circle C_0 contains circle C_1 but not C_2.