Department of EECS
York University

Term: Winter 2019
Instructor: Andy Mirzaian

# EECS2011: Fundamentals of Data Structures Sections M and Z

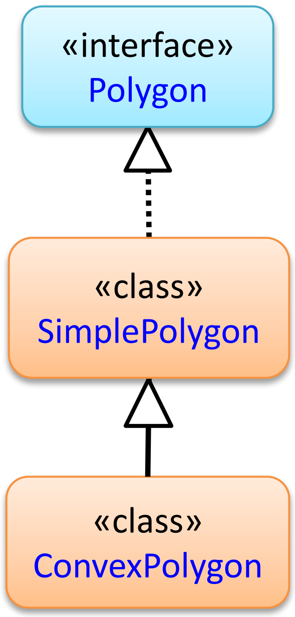
## Assignment 1

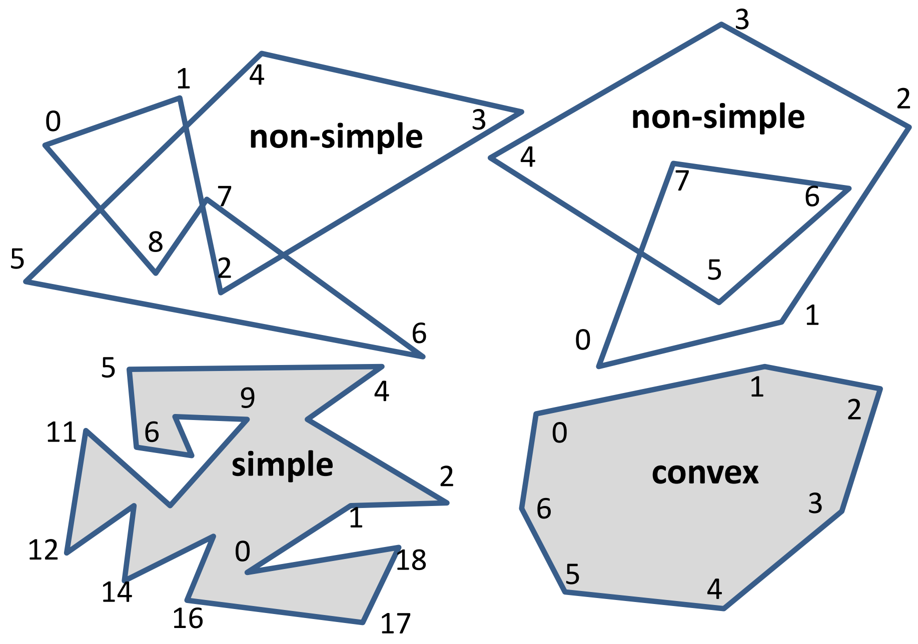
**Due: 10 pm, Friday, January 25, 2019**

- **Read the course FAQ on how to submit assignments electronically.**
- **Print your name, eecs account, and student ID # on top of every file you submit.**

**This assignment is about the polygon hierarchy shown in the figure below.**

We use **double** precision coordinates for vertices of the polygon. So, our Polygon **interface** is different from the java.awt.Polygon **class** in the Java API (which uses **int**-coordinate vertices) .





- Page 2 gives some preliminary descriptions.
- Page 3 shows some useful facts that you may use.
- Page 4 shows a summary of the interface and classes you are assigned to build and test.
  You may use additional types or members as you deem necessary in your design.
- Page 5 gives a small sample of input polygons you should test your program against.
  You should also use a number of other, larger, test cases of your own.
- Page 5 also lists what files you should submit.
- Page 6 describes an extra credit and optional work.

Click here to see the **code templates** and their **Javadocs**.

**Preliminaries:**
An $n$-sided polygon $P$ $(n \geq 3)$ is a cyclic sequence $(v_0, v_1, \cdots, v_{n-1})$ of vertices as we walk around the polygon boundary. Each vertex $v_i$ (the $i^{\text{th}}$ vertex) is a point in the plane represented by its **double** $x$ and $y$ coordinates. The line-segment $e_i$ between vertex $v_i$ and the next vertex $v_{(i+1)\ mod\ n}$ (in cyclic order around the boundary) is called the $i^{\text{th}}$ edge of $P$, for $i = 0..n-1$.

You may use the *Point2D.Double* class in the java.awt.geom package of the Java API to represent polygon vertices.

The polygon is said to be **simple** if no two non-adjacent pair of edges intersect. That is, two edges $e_i$ and $e_j$ are completely disjoint from each other whenever $1 < j - i < n - 1$.

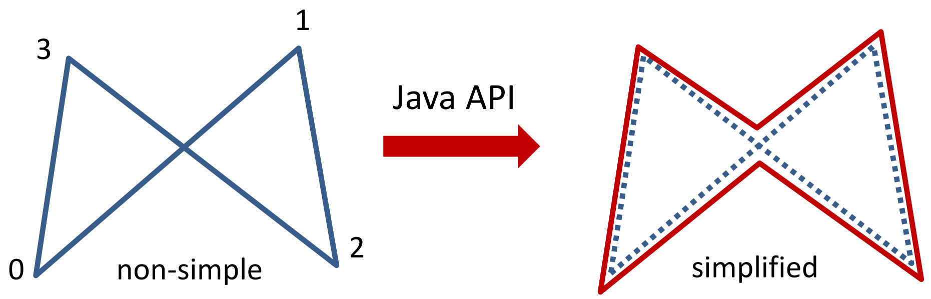
A simple polygon is said to be **convex** if the internal angle of every vertex is at most 180°. Equivalently, the simple polygon is convex if every turn is consistently in the same orientation (not clockwise or not counter-clockwise) as we walk around the polygon boundary. This latter condition is computationally more useful (see the description of the Delta Test on the next page).

The main methods we are interested in is polygon **perimeter** and **area**. Note that polygon area may not be well defined if the polygon is non-simple, since the notion of polygon "interior" may not be well defined in that case. We obviously need the boolean methods **isSimple** and **isConvex** as well.

**Extra Credit and Optional work:**
The last page talks about the **contains(p)** method which determines whether the polygon contains the given point p inside. The precondition is that the polygon is simple (otherwise, "inside" may not be well defined). This method can be implemented more efficiently on convex polygons than on simple polygons. Some hints are provided on how to implement this method.

**Remark:** There are some differences between our notions of polygon "interior" and "contains" and that of the Java API (especially when the polygon is non-simple). In the latter, Polygon implements the Shape interface which redefines the shape boundary as a path sequence, different from the vertex sequence. It implicitly adds edge crossings as new double vertices in the path sequence, and redirects the sequence to "simplify" the shape boundary. See the illustrative figure below. It uses the so called "winding number" to determine the "interior" of the shape.

**In short, you should follow our definitions.** ♦



**Useful Facts:**

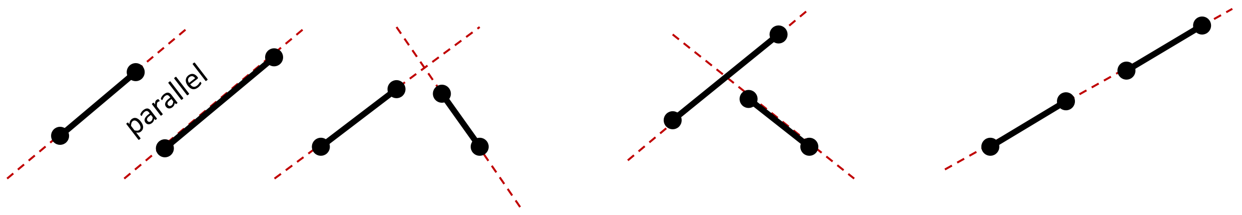
- **Delta Test:**
  Suppose we are given an ordered sequence of three points $(a, b, c)$; each point given by its $x$ and $y$ coordinates (e.g., $x_a$ and $y_a$). We want to know what is the orientation as we go from $a$ to $b$ to $c$ and back to $a$: is it clockwise (i.e., right turn), counter-clockwise (i.e., left turn), or collinear? The answer is given by the determinant of the $3 \times 3$ matrix shown below:

$$delta(a, b, c) = det \begin{bmatrix} x_a & y_a & 1 \\ x_b & y_b & 1 \\ x_c & y_c & 1 \end{bmatrix}$$

  This expression can be computed in $O(1)$ time with arithmetic operations on the xy-coordinates of the three given points and is a very useful quantity with many geometric applications. (It is analogous to the *compareTo* method on *Comparable* types.) So, it's worth providing a static helper method to compute $delta(a, b, c)$.

- **Triangle Signed Area:**
  The *signed area* of the oriented triangle $(a, b, c)$ is $\frac{1}{2} delta(a, b, c)$, where the sign is positive, negative, or zero if the orientation is counter-clockwise, clockwise, or collinear, respectively.

- **Line-Segment Disjointness Test:**
  A line-segment can be represented by a pair of delimiting points. We want to know whether a given pair of (closed) line segments $(a, b)$ and $(c, d)$ are disjoint. The possible cases are depicted in the figure below.



  This test can be done in $O(1)$ time (e.g., using the Delta Test). (How?) You would need repeated use of such a test in the polygon simplicity checking method. So, it's worth providing a static helper method for line-segment disjointness test. ***Note:*** *You must implement this method yourself. You are not allowed to use the line intersection method provided by the Java API. Exercise your logical and analytical thinking and problem solving skills.*

- **Polygon Area:**
  Consider the simple polygon $(v_0, v_1, \cdots, v_{n-1})$, where we denote the xy-coordinates of vertex $v_i$ by the pair $(x_i, y_i)$. For notational simplicity, we assume $v_n \equiv v_0$ and $v_{-1} \equiv v_{n-1}$ (i.e., index arithmetic is modulo n). Then the area of the simple polygon is:

$$\frac{1}{2} \left| \sum_{i=0}^{n-1} delta(o, v_i, v_{i+1}) \right| = \frac{1}{2} \left| \sum_{i=0}^{n-1} x_i (y_{i+1} - y_{i-1}) \right| ,$$

  where $o = (0,0)$ denotes the origin.
  (**Note:** it's absolute value of the sum, not the sum of absolute values!)
  This expression can be evaluated in $O(n)$ time using arithmetic operations.

**Interface Polygon:**

| | |
|---|---|
| **getSize()** | returns n, the number of edges of the polygon. |
| **getVertex(i)** | returns the $i^{\text{th}}$ vertex of the polygon with precondition $0 \leq i < n$. Throws *IndexOutOfBoundsException* if the precondition is violated. |
| **perimeter()** | returns the sum of the lengths of the edges of the polygon. |
| **area()** | returns area of the interior of the polygon if this notion is well defined; throws an exception if it's not. |

**Class SimplePolygon (implements Polygon):**

| | |
|---|---|
| **getNewPoly()** | constructs & returns a polygon, initialized by user provided data in O(n) time. |
| **toString()** | returns a String representation of the polygon in O(n) time. |
| **delta(a,b,c)** | returns twice the signed area of oriented triangle (a,b,c) in O(1) time. |
| **disjointSegments (a,b,c,d)** | returns true iff closed line-segments $(a, b)$ and $(c, d)$ are disjoint. Runs in $O(1)$ time. |
| **disjointEdges(i, j)** | returns true iff edges $e_i$ and $e_j$ of the polygon are disjoint. Runs in $O(1)$ time. |
| **isSimple()** | returns true iff the polygon is simple. Running time is $O(n^2)$. |
| **area()** | returns area of the interior of the polygon with precondition that the polygon is simple. Throws *NonSimplePolygonException* if the polygon is not simple (since in that case the polygon "interior" may not be well defined). Runs in $O(n)$ time, not counting the simplicity test. |

**Class ConvexPolygon (extends SimplePolygon):**

| | |
|---|---|
| **isConvex()** | returns true iff the polygon is convex, with precondition that the polygon is simple. This method runs in $O(n)$ time. If the polygon is non-simple, the correctness of the returned result is not guaranteed. |

**Class NonSimplePolygonException (extends Exception):**

Thrown to indicate that the polygon is non-simple.

**Class PolygonTester:**

This class has a main method that allows the user to input a variety of polygons and thoroughly test all aspects of the above types and methods, and displays or logs informative input-output.

**Sample polygon input format:**

Below are some sample input polygons listed by the xy-coordinates of the vertices in sequence around the polygon boundary. You should test your program against these samples. You should also test it against many other larger and carefully chosen polygons as well.

[ Format: n, followed by xy-coordinates of n boundary vertices in sequence. ]

Poly1: 5 8.9 21.8 29.1 8.8 39.2 20.3 14 11 28 25

Poly2: 7 28 2 31 5 28 10 14 14 5 10 8 4 18 1

Poly3: 9 6 10 20 3 23 3 23 8 27 3 30 3 20 15 16 5 20 14

Poly4: 13 5 6 13 2 12 6 20 2 16 12 17 11 19 5 13 11 19 15 8 12 14 7 5 11 9 6

Poly5: 13 5 6 13 2 12 6 20 2 18 12 17 11 19 5 13 11 19 15 8 12 14 7 5 11 9 6

Poly6: 22 14 7 15 8 17 7 17 5 15 6 14 4 12 6 11 9 15 11 7 12 8 11 7 9 10 11 8 6 10 5
11 3 16 3 18 4 19 8 16 9 14 9 13 8

Poly7: 4 6 1 9 5 5 8 2 4

**What files to submit:**

- **Polygon.java**
- **SimplePolygon.java**
- **ConvexPolygon.java**
- **NonSimplePolygonException.java**
- **PolygonTester.java**
- **TestIO.txt**
  This text file records the Input/Output results of your test cases. Use copy-&-paste from the Java program console to this text file. It should clearly show that you have tested at least 7 different polygons (non-simple, simple, convex), and for each of them you have thoroughly tested the corresponding class methods.

## Extra Credit and Optional:

Implement the new boolean instance method **contains(p)** in the SimplePolygon class **.** This method returns true iff the simple polygon contains the given point p in its interior or on its boundary. This method should take O(n) time, with the precondition that the polygon is simple (otherwise, "interior" may not be well defined).

**Note:** you are not allowed to use the Polygon.contains() method provided by Java API. We expect you to implemented this method yourself.

This method is now **inherited** by the ConvexPolygon class.
Now you **override** that method in the ConvexPolygon class, so that it runs in O(log n) time.

**Hints on how to implement this method:**

- *SimplePolygon.contains(p):*
  Conceptually shoot a ray emanating from point p in a direction of your choice (e.g., in the direction of the x-coordinate axis) and count how many times does that ray "cross" the boundary of the polygon. If that count is odd, then p is inside. If that count is even, then p is outside the polygon. If point p lies on any edge or vertex, then it is considered inside too.

  You can treat that ray as a sufficiently long line-segment, and use the disjointSegments test against edges of the polygon to count the number of crossings. Be careful to count properly if the ray passes through a vertex or is aligned with an edge of the polygon. Resolve the ambiguity by conceptually perturbing the ray with a slight angular rotation so that it does not pass through any vertex of the polygon.

- *ConvexPolygon.contains(p):*
  Can be done by a rotational binary search on the vertex sequence around the convex polygon boundary (using the Delta Test for orientation) …