Com S 228
Fall 2015
Exam 2

DO NOT OPEN THIS EXAM UNTIL INSTRUCTED TO DO SO

Name: ____

ISU NetID (username): ____

Recitation section **(please circle one):**

1.	M	11:00 am	(Alex, Kevin)
2.	M	10:00 am	(Caleb, Hyeyoung)
3.	M	4:10 pm	(Monica, John)
4.	T	11:00 am	(Ryan Y, Logan)
5.	T	10:00 am	(Blake, Kelsey)
6.	T	4:10 pm	(Andrew, Ryan K)

Closed book/notes, no electronic devices, no headphones. Time limit **75 minutes**.

Partial credit may be given for partially correct solutions.

- Use correct Java syntax for writing code.
- You are not required to write comments for your code; however, brief comments may help make your intention clear in case your code is incorrect.

If you have questions, please ask!

Question	Points	Your Score
1	28	
2	12	
3	18	
4	10	
5	32	
Total	100	

1. (28 pts) The `main()` method below executes a code snippet after the initialization of a `List` object. On the next page, you will see several snippets of code, each to be executed **within a separate call** of the `main()` method.

```
public static void main(String[] args) throws NoSuchElementException,
                                              IllegalStateException
{
      List<String> aList;
      ListIterator<String> iter, iter2;

      // initialization
      aList = new ArrayList<String>();
      aList.add("A");
      aList.add("B");
      aList.add("C");
      aList.add("D");
      aList.add("E");

      // code snippet
      // ...
}
```

Note that each snippet is *separate* and executed *independently* right after the initialization.

For each snippet,

a) show what **the output** from the `println` statement is, if any, and

b) draw **the state of `aList` and the iterator** after the code executes, and

c) do **not** display any other list that may appear in the code.

However, if the code throws an exception, do **not** draw the list but instead write down the exception that is thrown. In this case, **also show the output**, if any.

Use a bar (**|**) symbol to indicate the iterator's logical cursor position. For example, right after the statement

```
iter = aList.listIterator();
```

the list would be drawn as follows.

```
|A B C D E
```

(the first one has been done for you as an example). If two iterators appear in the code, **label** (or draw or color) **them** differently.

Suggestion: For **partial credit**, you may also want to draw the intermediate states of the list and iterator after executing every one or few lines of code in a snippet.

Code snippet	**Output**	**List and iterator state, or exception thrown**
`iter = aList.listIterator();`	(none)	|A B C D E

`// 3 pts`
`iter = aList.listIterator(aList.size());`
`iter.previous();`
`System.out.println(iter.previousIndex());`		

`// 4 pts`
`iter = aList.listIterator();`
`while (iter.hasNext()) {`
`    iter.next();`
`    if (iter.hasNext())`
`        System.out.println(iter.next());`
`}`		

`// 3 pts`
`aList.add("F");`
`aList.remove("C");`		

`// 5 pts`
`aList.add("O");`
`iter = aList.listIterator(aList.size() - 2);`
`iter.previous();`
`iter.set("X");`
`iter.set("Y");`
`iter.add("Z");`
`iter.remove();`		

`// 5 pts`
`iter = aList.listIterator(aList.size() / 2);`
`while (iter.hasNext()) {`
`    iter.next();`
`    iter.previous();`
`    iter.previous();`
`}`		

`// 8 pts`
`iter = aList.listIterator();`
`iter2 = aList.listIterator(aList.size());`
`while (iter.nextIndex() < iter2.previousIndex()) {`
`    String s = iter.next();`
`    String t = iter2.previous();`
`    if (s.compareTo(t) < 0) {`
`        iter.set(t);`
`        iter2.set(s);`
`    }`
`}`
System.*out*.println(iter.nextIndex() ==
`iter2.previousIndex());`
`iter.next();`
`iter.add("F");`
`iter.add("G");`		

2. (12 pts) For the next questions assume a doubly-linked list implementation of the List interface that includes a tail reference (as in the `DoublyLinkedList` example discussed in class, for instance). Assume that this includes an implementation of all methods of the `ListIterator` interface. The list has n elements.

a) (4 pts) Give the big-O time complexity of the following **List** API operation.

```
public boolean isEmpty()
```

b) (4 pts) Give the big-O time complexity of the following **List** API operation.

```
public void add(int index, E item)
```

c) (4 pts) Give the big-O time complexity of the following **ListIterator** API operation.

```
public void add(E item)
```

3. (18 pts) Infix and postfix expressions.

a) (9 pts) Convert the following infix expression into postfix.

```
2 * b % 3 + (a – (b – (c – d * e))) ^ (4 + b * (c – d)) ^ (1 – 6 / (i – j))
```

Postfix:

b) (9 pts) Convert the following postfix expression into infix.

```
5 f + 2 a – / b c d e + ^ ^ * 10 2 g h – a b + / * + –
```

Infix:

4. (10 pts) Calculate the greatest common divisor of 748 and 286 using the Euclidean algorithm.

5. (32 pts) You are provided an abstract class Shape and its subclass Point as below.

a) (9 pts) Implement the compareTo() method according to the Javadoc right above it.

```
public abstract class Shape implements Comparable<Shape>
{
    // coordinates of the reference point on the shape
    protected int x;
    protected int y;

    public Shape(int x, int y)
    {
      this.x = x;
      this.y = y;
    }

    /**
     * Compare this shape with a second shape s using their x coordinates, and
     * in the case of a tie, using their y coordinates.
     */
    public int compareTo(Shape s)
    {




    }
}
```

```
public class Point extends Shape
{
    private String color;

    public Point(int x, int y, String c)
    {
        super(x, y);
        color = c;
    }

    public int getX()
    {
        return x;
    }

    public int getY()
    {
        return y;
    }

    public String getColor()
    {
         return color;
    }

    /**
     * Output a point in the standard form (x, y).
     */
     @Override
    public String toString() {
         return "(" + x + ", " + y + ")";
    }
}
```

b) (5 pts) Suppose we are implementing a static method `selectionSort()` which performs selection sort on an array of objects of some class T. Fill a **wildcard** type in the blank preceding the method `selectionSort()` below so it can sort an array `pts[]` of `Point` objects by making use of the `compareTo()` method provided by the class `Shape`, as carried out in the `main()` method.

```
public class SortPoints
{
    public static <___________________________________________________________>
           void selectionSort(T[] arr)
    {
        // implementation details omitted
       // ...
    }


    public static void main(String[] args)
    {
        // initialize an array pts[] of Point objects
        // ...

        selectionSort(pts);
        System.out.println(Arrays.toString(pts));
    }
}
```

c) (18 pts) The method `GrahamScan()`, inside the class `ConvexHull` below, implements the algorithm Graham's Scan which constructs the convex hull of points from the array `pointsToScan[]`. Pay attention to the following:

1. You do **not** need to implement other methods in the `ConvexHull` class than `GrahamScan()`.

2. Within `GrahamScan()`, you **need only implement the actual scan**, not the preprocessing steps which are outlined in the preceding comments.

3. The algorithm uses a stack `vertexStack`. The **only** stack methods you may need are `push()`, `pop()`, `size()`, and `isEmpty()`.

4. You also need a method `leftTurn()` given below. Please read its Javadoc carefully.

```
public class ConvexHull {

    // Data structures
    // ...
    private Point[] pointsToScan;          // points operated on by Graham's scan
    private Point[] hullVertices;          // vertices of the constructed convex hull
                                           // in the counterclockwise order

    private PureStack<Point> vertexStack;  // stack used by Graham's scan

    //...

    /**
     * @return  true   if straight movements from p1 to p2 and then from p2 to p3
     *                  make a left turn at p2
     *          false  otherwise
     */
    private boolean leftTurn(Point p1, Point p2, Point p3)
    {
        // implementation details omitted
        // ...
    }


    public void GrahamScan()
    {
        // The following are four preprocessing steps that have been implemented
        // already. Do not write code for them.
        //
        // 1. Find the lowest point in the input array of Point objects.  In case of
        //    a tie, pick the leftmost one.
        // ...

        // 2. Remove duplicate points.
        // ...

        // 3. Sort the remaining points in increasing polar angle, and in case of
        //    a tie, in increasing distance to the lowest point.
        // ...

        // 4. Store the sorted sequence in the array pointsToScan[].
        // ...


        vertexStack = new ArrayBasedStack<Point>();

        // Convex hull of one point.
        if (pointsToScan.length == 1)
        {
            //  Create hullVertices[] to store the hull.
            // insert code below (2 pts)









            return;
        }


        // Convex hull of two points.
        if (pointsToScan.length == 2)
        {

            // insert code below (3 pts)













            return;
        }
```

```
// Initialize vertexStack by pushing the first three points
// from pointsToScan[] onto the stack.

// insert code below (3 pts)









// Scan the remaining points.
for (int i = 3; i < pointsToScan.length; ++i)
{
    // Get the next point from pointsToScan[].
    // Fill in the blank in the assignment below.


    Point pi = ________________________________________;  // (1 pts)

    while (true)
    {
        Point top = vertexStack.pop();
        Point nextToTop = vertexStack.peek();


        if (leftTurn(nextToTop, top, pi))
        {
            // insert code below (3 pts)






            break;
        }
    }

    // Push the point pi onto the stack.
    // insert code below (2 pts)






}
```

```
        // Extract the points from vertexStack and store them as vertices
        // in the array hullVertices[], where they will form a counterclockwise
        // traversal of the convex hull as the index increases.

        hullVertices = new Point[vertexStack.size()];
        int i = vertexStack.size() - 1;

        // insert a condition for the while statement below (2 pts)

        while (____________________________________________________________)
        {
            // insert code below (2 pts)







        }
    }

    // other methods of the class ConvexHull
    // ...
}
```

Appendix A: Excerpt from List documentation, for reference.

Method Summary	
boolean	**add**(E e) Appends the specified element to the end of this list. Returns true if the element is added.
void	**add**(int index, E element) Inserts the element at the specified position in this list. Throws IndexOutOfBoundsException if index is less than zero or greater than size().
void	**clear**() Removes all of the elements from this list.
E	**get**(int index) Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the index is less than zero or is greater than or equal to the size of the list.
int	**indexOf**(Object obj) Returns index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**() Returns true if this list contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this list in proper sequence.
ListIterator<E>	**listIterator**() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	**listIterator**(int index) Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position . Throws IndexOutOfBoundsException if the index is less than zero or is greater than size.
E	**remove**(int index) Removes and returns the element at the specified position in this list. Throws IndexOutOfBoundsException if index is less than zero or is greater than or equal to size of the list.
boolean	**remove**(Object obj) Removes the first occurrence of the specified element from this list, if it is present. Returns true if the list is modified.
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element. Throws IndexOutOfBoundsException if index is less than zero or is greater than or equal to the size()
int	**size**() Returns the number of elements in this list.

Excerpt from Iterator documentation, for reference.

Method Summary	
boolean	**hasNext**() Returns true if the iteration has more elements.
E	**next**() Returns the next element in the iteration. Throws NoSuchElementException if there are no more elements in the collection.
void	**remove**() Removes from the underlying collection the last element returned by next(). Throws IllegalStateException if the operation cannot be performed.

Excerpt from Collection documentation, for reference.

Method Summary

boolean	add(E e) Ensures that this collection contains the specified element (optional operation).
void	clear() Removes all of the elements from this collection (optional operation).
boolean	contains(Object obj) Returns true if this collection contains the specified element.
boolean	isEmpty() Returns true if this collection contains no elements.
Iterator<E>	iterator() Returns an iterator over the elements in this collection.
boolean	remove(Object o) Removes a single instance of the specified element from this collection, if it is present
int	size() Returns the number of elements in this collection.

Excerpt from ListIterator documentation, for reference.

Method Summary

void	add(E e) Inserts the specified element into the list.
boolean	hasNext() Returns `true` if this list iterator has more elements in the forward direction.
boolean	hasPrevious() Returns `true` if this list iterator has more elements in the reverse direction.
E	next() Returns the next element in the list. Throws `NoSuchElementException` if there are no more elements in the forward direction.
int	nextIndex() Returns the index of the element that would be returned by `next()`.
E	previous() Returns the previous element in the list. Throws `NoSuchElementException` if there are no more elements in the reverse direction.
int	previousIndex() Returns the index of the element that would be returned by `previous()`.
void	remove() Removes from the list the last element that was returned by `next` or `previous`. Throws `IllegalStateException` if the operation cannot be performed.
void	set(E e) Replaces the last element returned by `next` or `previous` with the specified element. Throws `IllegalStateException` if the operation cannot be performed.

(this page is for scratch only)

(scratch only)

(scratch only)

(scratch only)

(scratch only)

(scratch only)