CS 201, Fall 2015 — Oct 23rd — Exam 2 Name: Solution

Question 1. [5 points] State a big-O upper bound on the worst case running time of the given method, where the problem size N is the number of elements in the array passed as its parameter. Explain your answer briefly.

```
public static int mystery(int[] arr) {
  int sum = 0;                                          - O(1)

  for (int i = 0; i < arr.length*arr.length; i++) {     - N^2 iterations
    sum += arr[i % arr.length];                         - O(1)
  }

  return sum;                                           - O(1)
}
```

$N^2 \cdot O(1) + O(1) + O(1)$ is $O(N^2)$

Question 2. [5 points] State a big-O upper bound on the worst case running time of the given method, where the problem size N is the number of elements in the array passed as its parameter. Explain your answer briefly.

```
public static int mystery(int[] arr) {
  int sum = 0;                                          - O(1)

  for (int i = 0; i < arr.length*arr.length; i++) {     - N^2 iterations
    sum += arr[i % arr.length];                         - O(1)
  }

  for (int i = 0; i < arr.length; i++) {                - N iterations
    sum += arr[i];                                      - O(1)
  }

  return sum;                                           - O(1)
}
```

$N^2 \cdot O(1) + N \cdot O(1) + O(1) + O(1)$

is $O(N^2)$

Question 3. [5 points] State a big-O upper bound on the worst case running time of the given method, where the problem size N is the number of elements in the array passed as its parameter. Explain your answer briefly.

```
public static int mystery(int[] arr) {
  int sum = 0;                                          ~O(1)

  for (int i = 0; i < arr.length; i++) {                — N iterations
    for (int j = 0; j < arr.length; j++) {              — N iterations
      sum += arr[(i*j) % arr.length];                   — O(1)
    }
  }

  return sum;                                           ~O(1)
}
```

N · N · O(1) + O(1) + O(1) is $O(N^2)$

Question 4. [5 points] State a big-O upper bound on the worst case running time of the given method, where the problem size N is the number of elements in the array passed as its parameter. Explain your answer briefly.

```
public static int mystery(int[] arr) {
  int sum = 0;                                          —O(1)

  for (int i = 0; i < arr.length; i++) {                — N iterations
    for (int j = i; j >= i; j--) {                      —— always exactly 1 iteration!
      sum += arr[(i*j) % arr.length];                   — O(1)
    }
  }

  return sum;                                           —O(1)
}
```

N · 1 · O(1) + O(1) + O(1) is $O(N)$

Question 5. [5 points] State a big-O upper bound on the worst case running time of the given method, where the problem size N is the value of the method's parameter. Explain your answer briefly.

```
public static int mystery(int n) {
  int sum = 0;

  for (int i = n; i > 0; i = i / 2) {
    sum += i;
  }

  return sum;
}
```

— k iterations (see below)

$O(\log n)$

assume $n = 2^k$

number of times n can be divided by 2 before reaching 1 is k.

Solving for k, take base 2 log of both sides, $\log_2 n = k$

Question 6. [5 points] Briefly explain the problem with the following method, and how to fix it.

```
public static<E> int countGreaterThan(ArrayList<E> list, E value) {
  int count = 0;

  for (int i = 0; i < list.size(); i++) {
    E elt = list.get(i);
    if (elt.compareTo(value) > 0) {
      count++;
    }
  }

  return count;
}
```

E is not guaranteed to implement Comparable, so we can't call compareTo on an object of type E

Solution: declare as

public static < E extends Comparable<E> > ...

(Let Comparable<E> be an upper bound type for E.)

Question 7. [10 points] Complete the following method, called `makeAllPositive`. It takes a reference to an `ArrayList` of `Integer` elements as a parameter. It should change all of the negative elements in the list to positive values. Example JUnit test:

```
ArrayList<Integer> a = new ArrayList<Integer>();
a.addAll(Arrays.asList(-9, 0, -4, -2, 4));
makeAllPositive(a);
assertEquals((Integer)9, a.get(0));
assertEquals((Integer)0, a.get(1));
assertEquals((Integer)4, a.get(2));
assertEquals((Integer)2, a.get(3));
assertEquals((Integer)4, a.get(4));
```

Note that the Java compiler will automatically convert between `int` and `Integer` values.

Hints:

- Use the `size` method to get the number of elements in the list
- Use the `get` method to retrieve the value at a specific index
- Use the `set` method to change the value at a specific index

```
public static void makeAllPositive(ArrayList<Integer> list) {
    for (int i=0; i<list.size(); i++) {
        Integer n = list.get(i);
        if (n < 0) {
            list.set(i, -n);
        }
    }
}
```

Question 8. [10 points] Construct the class `Shape` that has abstract methods `calcPerimeter` and `calcArea`, which have no parameters, and return floating point numbers. `Shape` also has fields named `type`, `origin`, `perimeter`, and `area`. `type` is a `String`, `origin` is a `Point`, and `perimeter` and `area` are floating point numbers.

Make sure to define all of the appropriate accessor functions for the fields in `Shape`. Allow only the constructor to set `type`, and for extra credit (+2), only allow sub-classes of `Shape` to calculate or change `perimeter` and `area`.

```
public abstract class Shape {
    private String type;
    private Point origin;
    private double perimiter, area;

    public Shape(String type, Point origin){
        this.type = type;
        this.origin = origin;
    }
    public String getType() { return type; }
    public Point getOrigin() { return origin; }

    public abstract double calcPerimeter();
    public abstract double calcArea();

    protected void setPerimeter(double p) { perimeter = p; }
    protected void setArea(double a) { area = a; }

    public void setOrigin(Point o) { origin = o; }
    public double getArea() { return area; }
    public double getPerimeter() { return perimeter; }
}
```

Question 9. [10 points] Create the concrete class `RegularPolygon` from as a subclass of the `Shape` class you specified in Question 8. `RegularPolygon` should also implement the `Comparable` interface, comparing the areas of the two objects involved in the comparison. It has a constructor that accepts values for `type`, `origin`, `sides`, and `length`. The constructor calls `calcPerimeter` and `calcArea` to initialize the `perimeter` and `area` fields.

Implement the appropriate accessor methods, restricting access so that only the constructor can set the `side`, `length`, `perimeter`, and `area` fields. Remember to declare and implement everything necessary to make `RegularPolygon` a concrete class, except that you can insert "`<CODE>`" in the bodies of any required methods that are not accessor methods or part of the `Comparable` interface.

```
public class RegularPolygon extends Shape implements Comparable<RegularPolygon>{
  private int sides;
  private double length;

  public RegularPolygon(String type, Point origin, int sides, double length) {
      super(type, origin);
      this.sides = sides;
      this.length = length;
      setArea(calcArea());
      setPerimiter(calcPerimeter());
  }

  public double calcPerimeter() { <code> }
  public double calcArea() { <code> }

  public int getSides() { return sides; }
  public double getLength() { return length; }

  public int compareTo(RegularPolygon p) {
      if(this.getArea() < p.getArea()) { return -1; }
      else if(this.getArea() > p.getArea()) { return 1; }
      else return 0;
  }
}
```