NAME: ___________________________________________ ____________________________

Answer all questions in the space provided. Use the backs of the pages if you need extra space. You may have a single-page one-side crib sheet for this exam.

If a question asks for code and you’re not sure how to write the code, write as much as you can, with comments to explain the part you don’t know how to write. It is better to write something showing what you know than to leave questions blank.

There are no deliberate errors in the code on this exam.

If you do not understand a problem, make an assumption, write it down, and answer the problem based on your assumption. If your assumption is reasonable, you will receive full credit. Reasonable assumptions do not make the problem significantly easier than intended.

Please make sure that you have all 11 pages before beginning the test.

If you are stuck on a question, move on and come back to it later.

<table>
<thead>
<tr>
<th>No.</th>
<th>Pts.</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>
1. (20 points) In this question, you will develop and test a program for working with ArrayLists of integers.

a. Fill in the blanks in the following method that finds the maximum value in an ArrayList of Integers

public int getMaximum(ArrayList<Integer> numbers)
{
    int first = numbers.get(0); // first is the first number in the list

    int max = first; /* value of first integer */

    for (int p = 1; p < numbers.size(); p++)
    {
        int current = numbers.get(p); /* current number in the list */
        if (current > max)
        {
            max = current; /* new value for max */
        }
    }

    return max;
}
b. Modify the method from part a so that it modifies the ArrayList by moving the maximum element from the array to the end. Any incorrect code that is identical to part a will not be penalized again. (Hint: to move an element, remove it from its old position, and insert it into its new position).

```java
Public ArrayList<Integer> moveMaxToEnd (ArrayList<Integer> numbers)
{
    int maxLocation = 0;
    int first = numbers.get(0);
    int max = first;
    for (int p = 1; p < numbers.size(); p++)
    {
        int current = numbers.get(p);
        if (current > max)
        {
            max = current;
            maxLocation = p;
        }
    }
    numbers.remove(maxLocation);
    numbers.add(max);  //inserts at end
    return numbers;
}
```

c. Explain how your code takes advantage of autoboxing and/or autounboxing.

In a statement like `int first = numbers.get(0)`, autounboxing changes the Integer in the arraylist to an int. In a statement like `numbers.add(max)`, autoboxing changes the int max to an Integer before adding to the arraylist.
2. (10 points) Write a recursive method “int sumodd (int m)” that returns the sum of all odd numbers up to and including m. Examples: sumodd(3) returns 4 (1+3), sumodd(5) returns 9 (1+3+5), and sumodd(6) also returns 9 (1+3+5).

    int sumodd(int m){
      if(m <1) return 0; //base case
      if(m%2 == 0) //even number
        return sumodd(m-1); //use next lower odd number
      else
        return m + sumodd(m-2); //add m to sum of smaller odd numbers
    }
3. (15 points) Consider the following code that implements a recursive MergeSort on arrays:

```java
public static void mergeSort(double[] theArray, int first, int last) {
    // Part B1
    if (first >= last) return;
    // Part B2
    else if (last == 1 + first) {
        if (theArray[first] > theArray[last]) {
            double tmp = theArray[first];
            theArray[first] = theArray[last];
            theArray[last] = tmp;
        }
    } else {
        // Part B3
        int middle = (first + last) / 2;
        mergeSort (theArray, first, middle);
        mergeSort (theArray, middle + 1, last);
        // Part B4
        merge (theArray, first, last);
    }
}

public static void merge(double[] theArray, int first, int last) {
    // This method takes the part of the array between first and last, which
    // two pieces in sorted order, and merges them into one sorted list.
    // Example: theArray is { 1, 4, 3, 2, x, x, x, x, } (x’s are numbers that don’t
    // change). First is 0 and last is 3. After running this method, theArray is
    // {1, 2, 3, 4 , x, x, x, x}
}
```

a. Draw a picture, or explain in words, exactly what will happen when MergeSort is called with the array {4, 1, 2, 3, 5, 8, 4}, 0 and 6 as parameters. Be sure to show all (recursive) calls to MergeSort and the values of the parameters as well as the contents of the array when MergeSort returns.

I will list the calls to MergeSort by their parameters, using A instead of writing out the array each time.

The first call is MergeSort(A, 0, 6)

This calls MergeSort(A, 0, 3) and MergeSort(A, 4, 6)

MergeSort(A, 0, 3) calls MergeSort(A, 0, 1) and MergeSort(A, 2, 3)
MergeSort(A, 0, 1) is a base case and since A[0] (4) < A[1] (1) it switches 4 and 1


A is now {1, 4, 2, 3, 5, 8, 4}

Both recursive calls for MergeSort(A, 0, 3) are finished, so merge(A, 0, 3) is called.

A is now {1, 2, 3, 4, 5, 8, 4}

MergeSort(A, 4, 6) calls MergeSort(A, 4, 5) and MergeSort(A, 6, 6)


MergeSort(A, 6, 6) is a base case, and since first=last, nothing is changed.

Both recursive calls for MergeSort(A, 4, 6) are finished, so merge(A, 4, 6) is called.

A is now {1, 2, 3, 4, 4, 5, 8}  (sublists {5, 8} and {4} are merged to make {4, 5, 8})

b. A recursive method has one or more base cases, some code that takes a “single step”, and recursive calls that use the method on a smaller case. For each of the commented parts of the code (B1, B2, B3, and B4 -- comments are before the statements), identify which of the above items (base, step, recursive call) that code implements.

B1: A base case (one-element list)

B2: Another base case (two-element list)

B3: The three lines following this comment implement the recursive calls. Dividing the list by computing “mid” could be considered a single step.

B4: A single step (merging the lists that have been recursively sorted)

c. It is important to know that every recursive call is solving a smaller problem, i.e. a problem closer to the base case. Explain why the sequence of calls generated by this method will eventually end with a base case.

Each call to MergeSort has a list that is (approximately) half as big as the previous list. If you keep dividing a list in half, you will eventually end up with a list of size 1 or size 2, which are the base cases.
4. (20 points) Circle one or more phrases from the choices in brackets that correctly complete the statements.

a. Generally, implementing a search recursively means it does [ more / less / about the same number of ] comparisons as the non-recursive version of the same search.

b. [ Binary search / sequential search ] does not work correctly unless the list is sorted.

c. [ Binary search / sequential search ] generally does more comparisons than the other search.

d. In an unsorted list of 1000 elements, determining that an element is not in the list usually takes around [100 / 500 / 1000 / 2000 ] comparisons.

e. A(n) [ abstract class / implementation / inheritance / interface / static class ] can contain methods for which no code is specified.

f. A(n) [ abstract class / implementation / inheritance / interface / static class ] can contain methods that include code.

g. Methods that are not defined in the [subclass / superclass / base class / derived class ] can be inherited from the [ subclass / superclass / base class / derived class ]

h. In the phrase “public class name VERB another-class”, when another-class is an ordinary class, VERB should be [ inherits / extends / implements / abstracts ]

i. In the phrase “public class name VERB another-class”, when another-class is an interface, VERB should be [ inherits / extends / implements / abstracts ]

j. The aspect of object-oriented programming that allows which object’s method is called to be determined at runtime, based on the kind of object is called [ overloading / inheritance / polymorphism ]
5. (20 points) Suppose you are designing an aquarium simulation. Classes that you will use include: Drawable_Object (any object that can be drawn. Must have a paint() method and also has a getX() and getY() method). Moveable_Object (any drawable object that can also move. Has a paint() method, and getX(), getY(), setX(), setY()). Fish (a moveable object), RedFish (a red fish), Plant (a drawable object that is not a movable object), and SeaHorse (a moveable object).

a. Draw a diagram that shows the object hierarchy (as it would look if you defined these classes in BlueJ). Mark any classes that are actually interfaces or abstract classes.

b. Give one example of a method that should be inherited. What is the method, which class is it defined in, and which class inherits it?

getX() and getY() are defined in DrawableObject and inherited by everything else.

moveX() and moveY() are defined in MoveableObject and inherited by Fish, SeaHorse and RedFish

The version of paint() that is defined in Fish is inherited by RedFish
c. Name one abstract class or interface (there should be at least one) and explain why it is abstract or an interface.

DrawableObject and MoveableObject are both abstract because they have abstract paint() methods. Subclasses of these objects must implement a paint() method.

d. Write the first line (class declaration) for each of the classes: DrawableObject, MoveableObject, Fish, RedFish, Plant, and SeaHorse.

```java
public abstract class DrawableObject
public abstract class MoveableObject extends DrawableObject
public class Fish extends MoveableObject
public class RedFish extends Fish
public class Plant extends DrawableObject
public class SeaHorse extends MoveableObject
```
6. (15 points) Consider the following classes:

```java
public class X{
    private String name;
    protected int val;
    private int safeX;
    public X(){
        name = "X";
        val = 1;
    }
    public String getName(){
        return name;
    }
}

public class Y extends X {
    private int yval;
    protected int safeY;
    public Y(){
        name = "Y";
    }
}
```

a. For each of the print statements in the following code, state whether or not it is an error. If it is an error, explain the error. If it is not an error, tell what is printed.

```java
X myX = new X();
Y myY = new Y();
System.out.println(myX.val); //ok if it is in the same package as X -- prints 1
System.out.println(myX.getName()); //ok -- prints X
System.out.println(myX.yval); //error (X has no yval)
System.out.println(myY.val); //ok if it is in the same package as X&Y -- prints 1 (value inherited from X)
System.out.println(myY.getName()); //if name were protected, this would print Y. But the constructor as written won’t compile (see below)
System.out.println(myY.yval); //error -- yval is private, so inaccessible
```

b. Fill in the boxes of the following table with “yes” or “no”:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Can be accessed by a method in X</th>
<th>Can be accessed by a method in Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Yes</td>
<td>No -- actually y will not compile, because the constructor cannot access name</td>
</tr>
<tr>
<td>val</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>safeX</td>
<td>Yes</td>
<td>no</td>
</tr>
<tr>
<td>safeY</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
(extra page for scratch paper)