# CSE 331 Final Exam 6/7/16 **Sample Solution**

**Question 1.** (10 points, 1 each) Warmup. For each statement, circle T if it is true and F if it is false.

a) T / F Integration testing verifies the operation of a single specific module.

b) T / F A regression test compares the current output of a test with a previously known value.

c) T / F Constructors can invoke any methods without problems, even ones overridden in subclasses.

d) T / F A realistic test suite should aim for 100% path coverage.

e) T / F It is possible to use Java Interfaces to construct a Java subtype graph (i.e., graph of subtyping relationships) that does not form a strict hierarchical tree.

f) T / F A static factory method may return an object that has any subtype of the declared return type.

g) T / F A static factory method must return a new, unique object each time it is called.

h) T / F The Builder pattern is appropriate when constructing objects with many possible parameters.

i) T / F The Adapter pattern keeps functionality of an object the same but changes the interface presented to client code.

j) T / F In a Java Swing application, after a program updates data that is displayed on the screen, it should call `paintComponent` to have the screen redrawn.

**Question 2.** (9 points, 3 each) Short answer. You shouldn't need more than a couple of sentences for each answer. (i.e., please keep it short)

(a) In the model-view-controller (MVC) pattern it is possible to use either pull updating or push updating to send to the view new information from the model. When and why is it better to use pull updating instead of push updating?

**Pull updating is particularly appropriate if the viewer(s) need only part of the information contained in the model, especially if the model contains a great deal of data and a viewer only needs a small part of it. With pull updating each viewer can retrieve just the information that it needs, which might be different for different viewers.**

(continued on next page)

**Question 2 (cont.)** (b) If a method is called with inappropriate argument values, there are two possible ways to handle the situation. One is to specify that a particular exception will be thrown if this happens (`@throws`). The other is to specify a precondition (`@requires`) and, as a defensive programming measure, use an `assert` statement to terminate the program if the precondition is violated. When is it more appropriate to use the precondition (`@requires`) and `assert` instead of including a documented exception in the method specification?

**Preconditions and asserts are most appropriate for methods that are private or where clients of the method are known (such as other code developed by the same team). For methods that are part of a public API it is best to minimize or eliminate the use of preconditions and specify the method behavior for as many argument values as possible, throwing appropriate exceptions when needed to signal invalid argument values.**

**Note: It is true that if checking a requirement is too expensive (e.g., "the array is sorted" for binary search) we would normally use a precondition instead of a documented exception and would not include code to check the requirement. But the question asked about the tradeoff between using a precondition plus assert vs. documenting an exception, which means that the code to check the condition will be present in either case.**

(c) A debugging strategy that is often surprisingly effective is known as "rubber duck debugging". What is it, and what is the main reason that it works (when it does)?

**The phrase refers to the strategy of trying to debug code by explaining it to someone else – even if it's a rubber duck. The reason that it is effective is that trying to explain something often can lead to insights about why things are not working as expected.**

**Question 3**. (12 points) A Generic question. We have the following method, which returns the average of the numbers in a list of Integer values, or returns 0.0 if the list is empty:

```
public static double avg(ArrayList<Integer> nums) {
  if (nums.size() == 0)
    return 0.0;
  double sum = 0;
  for (int i = 0; i < nums.size(); i++) {
    sum += nums.get(i);
  }
  return sum / nums.size();
}
```

This method would be much more useful if it wasn't restricted to a particular collection type (`ArrayList`) and a single element type (`Integer`). Rewrite this method using generics and collections so that it can be used to compute the average value of the items in any `Collection` that contains any subtype(s) of `Number`.

Hint/reminder: `Number` is an abstract class, so it is not possible to create variables or to return values of type `Number`. However all of the arithmetic needed can be done using type `double` and using the `doubleValue()` method to convert any `Number` to a `Double`. Also, remember that `get(i)` is not supported by all `Collection` types.

```
public static <T extends Number>
double avg(Collection<T> nums) {
  if (nums.size() == 0)
    return 0.0;
  double sum = 0.0;
  for (T n: nums) {
    sum += n.doubleValue();
  }
  return sum / nums.size();
}
```

**Question 4.** (7 points) Bugs 'R Us. The following method is supposed to compute a reciprocal product as defined in the specification.

```
/**
 * @requires 0 <= x <= 10 (So no overflow or underflow bug)
 * @returns 0 if x or y is 0
 *          otherwise, return the product of the reciprocals
 *          of x and y (i.e if x is 5 and y is 2, return 0.1,
 *          which is 1/5 * 1/2)
 */
public static double getReciprocalProduct (int x, int y) {
  if (x == 0 || y == 0) {
    return 0;
  }
  double xRec = 1/x;
  double yRec = 1/y;
  return xRec * yRec;
}
```

Unfortunately it doesn't work.

(a) (2 points) What's the defect (i.e., the bug)?

**The code uses Java's truncating integer division to compute `xRec` and `yRec`, and those results will be 0 when either denominator is greater than 1.**

(b) (3 points) Write a test case that reproduces the bug by completing the JUnit method below. (Hint: this might be very short).

```
@Test
public void testReciprocalProductDivision() {
   assertEquals(0.1, getReciprocalProduct(5, 2));
}
```

**Note: since floating-point values are involved, it would be better to use the 3-argument version of `assertEquals` for `doubles` and give a delta tolerance for the comparison. But for this exam question, either version received full credit.**

(c) (2 points) Describe how to fix the bug (one sentence should be enough)

**Any change that forces `1/x` and `1/y` to be computed using floating-point arithmetic would work. Examples: `1.0/x` or `1/(double)x` or `(double)1/x`. But note that `(double)(1/x)` will not work since truncating integer division is still performed before the result is cast to `double`.**

The next several questions refer to the following `Point` class and several related subclasses. To save space, the standard `equals`, `hashCode`, and `toString` methods are omitted and their implementation details are not needed for the questions. We've also omitted most of the specification comments – the specifications are straightforward and match the code.

```
class Point {
  // rep: (x,y) coordinates on the plane
  // protected to give subclasses direct access
  protected float x, y;

  // constructor
  public Point(float x, float y) {
    this.x = x;  this.y = y;
  }

  // observers
  public float getX() { return x; }
  public float getY() { return y; }

  // = distance to origin
  public double distance() {
    return Math.sqrt(x*x + y*y);
  }
}
```

In other words, a `Point` represents an immutable 2-D point in the usual Cartesian coordinate space. Methods are provided to access the x and y coordinates of a `Point` and to compute its distance from the origin.

Now consider the following additional classes, which extend the original class `Point` directly or indirectly. These are all legal Java subclasses, and the code compiles without errors or warnings.

```
class MutablePoint extends Point {
  public MutablePoint(float x, float y) {
    super(x,y);
  }
  // mutators
  public void setX(float x) { this.x = x; }
  public void setY(float y) { this.y = y; }
}
```

(Continued on the next page. You may remove these pages for convenience if you wish.)

Additional subclasses to go with the code on the previous page.

```
class Point3D extends Point {
  // rep: add 3rd coordinate
  protected float z;

  // construct 3-D point at (x,y,z)
  public Point3D(float x, float y, float z) {
    super(x, y);
    this.z = z;
  }

  // observers
  public float getZ() { return z; }

  public double distance() {
    return Math.sqrt(x*x + y*y + z*z);
  }
}

class ColorPoint extends Point {
  // rep: add color
  protected java.awt.Color color;

  // construct 2-D point with given color
  public ColorPoint(float x, float y, java.awt.Color c) {
    super(x, y);
    this.color = c;
  }
  public java.awt.Color getColor() { return color; }

}

class ColorPoint3D extends ColorPoint {
  // rep: add 3rd coordinate
  protected float z;

 // construct 3-D point with given color
  public ColorPoint3D(float x,float y,float z,java.awt.Color c) {
    super(x, y, c);
    this.z = z;
  }

  // observers
  public float getZ() { return z; }

  public double distance() {
    return Math.sqrt(x*x + y*y + z*z);
  }
}
```

( Actual questions about all this code on the next pages. ☺ )

**Question 5.** (10 points) As mentioned above, these classes have the Java subtype relationships implied by the code. The code compiles with no errors or warnings. However it's not entirely clear which classes are proper true subtypes of each other.

For each of the types below, circle the names of all the other types that are a proper, true subtype of the given type, regardless of whether they are considered Java subtypes. You should assume that there is no trickery in any of the specifications for any of the classes – no hidden preconditions or exceptions or other unexpected things. Use the code to determine the behavior of each type. (We omitted the full JavaDoc specifications since it would have taken another 2 or 3 pages and would not have added any useful information to what is implied by the code – apologies for the implicit "specification by implementation" in this particular question.)

Every type is by definition a subtype of itself. That does not need to be included in the answers.

(a) The following types are true subtypes of Point (circle):

MutablePoint   Point3D   ColorPoint   ColorPoint3D

(b) The following types are true subtypes of MutablePoint (circle):

Point   Point3D   ColorPoint   ColorPoint3D

(c) The following types are true subtypes of Point3D (circle):

Point   MutablePoint   ColorPoint   ColorPoint3D

(d) The following types are true subtypes of ColorPoint (circle):

Point   MutablePoint   Point3D   ColorPoint3D

(e) The following types are true subtypes of ColorPoint3D (circle):

Point   MutablePoint   Point3D   ColorPoint

**This question turned out to have problems that weren't anticipated when we wrote it and that made the true subtyping properties not work as intended. The biggest problem is that since the distance calculation for a 3D point works differently than for a 2D point, that meant that two points that were equal when considered as 2D points could actually be different if one or both of them were 3D points. So even if they were immutable, they wouldn't have the proper true subtyping relationship. We made some adjustments during grading, and when using this question to study in future quarters, keep in mind that there are problems with it.**

**Question 6.** (9 points) We would now like to create a class `Rectangle` using `Points` as instance variables holding the coordinates of the `Rectangle`'s upper-left and lower-right corners. The upper-left corner *must* be above and to the left of the lower-right corner, i.e., the `Rectangle` must have width and height greater than 0.0.

Here is the beginning of Rectangle.java. You should fill in:

(a) The overview of the class in the comment above "class Rectangle",
(b) A suitable rep invariant, and
(c) A suitable abstraction function.

(Do not worry about putting "//" or similar comment symbols in front of every line.)

```
/** Class Rectangle - add overview here
  *
  * A Rectangle represents a rectangle on the plane.  It is
  * defined by two points, the upper-left and lower-right
  * corners.  A rectangle has positive area, or, in other
  * words, the upper-left corner must be above and to the
  * left of the lower-right corner.
  *
  *
  *
  *
  */
public class Rectangle {
  // rep: corners of the rectangle
  private Point ul;      // upper-left corner
  private Point lr;      // lower-right corner

  // rep invariant:

  ul != null and lr != null and
  ul.getX() < lr.getX() and ul.getY() > lr.getY()

  // abstraction function:

  A Rectangle is a 2-D rectangle where ul gives the coordinates
  of the upper-left corner and lr gives the coordinates of the
  lower-right corner.
```

**Question 7.** (8 points) Specification & implementation. Our `Rectangle` class needs a proper constructor to initialize the instance variables of a new `Rectangle` and to guarantee that the new `Rectangle` is properly formed (i.e., the new `Rectangle` satisfies its rep invariant).

Complete the JavaDoc specification and provide an appropriate implementation for the constructor for this class. The heading for the constructor with appropriate parameters is supplied below the JavaDoc comments. You need to supply the implementation and decide the most appropriate way to handle any error conditions. You should leave any unneeded parts in the JavaDoc comment blank.

```
/** (add overview here)
 *
 *  [Construct a new Rectangle with the given corners]
 *
 * @param ul upper-left corner of new Rectangle
 *
 * @param lr lower-right corner of new Rectangle
 *
 * @requires
 *
 * @modifies
 *
 * @effects Construct a new Rectangle with given corners
 *
 * @throws IllegalArgumentException if ul==null or
 *         lr==null or if the upper left corner is not
 *         above and to the left of the lower right one
 *
 * @returns
 */
public Rectangle(Point ul, Point lr) {
  if (ul == null || lr == null ||
      ul.getX() >= lr.getX() || ul.getY() <= lr.getY())
    throw new IllegalArgumentException();
  this.ul = ul;
  this.lr = lr;
}
```

**Notes: CSE 331 specifications use `@effects` to describe the results of a constructor, unlike the Java libraries which use the overview section. We did require that `@effects` be specified; we did not penalize code that omitted the overview and included the description only under `@effects`.**

**Some answers used `@requires` to specify the preconditions. That received almost full credit, but for a public constructor like this it is better to specify an appropriate exception so that the constructor is defined for as many input values as possible.**

**Question 8.** (10 points, 2 each) `hashCode`s. We will assume that our `Rectangle` class includes the following `equals` method:

```
@Override
public boolean equals(Object o) {
  if (! (o instanceof Rectangle))
    return false;
  Rectangle other = (Rectangle)o;
  return this.ul.equals(other.ul) && this.lr.equals(other.lr);
}
```

Here are several possible `hashCode` methods for Rectangle. For each one, circle OK if it is *guaranteed* to be a legal `hashCode` method for `Rectangle`, given the `equals` method above. Circle ERROR if it does not satisfy the contract for `hashCode`. All of these functions compile without errors or warnings.

Hint: we can't make any assumptions about how `Point`'s `equals` method works, other than it is consistent with `Point`'s `hashCode` method.

OK ERROR `int hashCode() { return 331; }`

OK ERROR `int hashCode() { return ul.hashCode(); }`

OK ERROR
```
int hashCode() { return ul.hashCode() +
                        lr.hashCode(); }
```

OK ERROR
```
int hashCode() { return
    (int)Math.max(ul.hashCode(), lr.hashCode()); }
```

OK ERROR
```
int hashCode() { return
                 (int)(ul.getX()+lr.getX()); }
```

**Note: the last part of this question was the Official Trick Question® on this exam. The "natural" `equals` for `Point` would be equality of x and y coordinates, in which case the last `hashCode` would be fine. But since we don't know how equality for `Points` is actually specified (it might be something like distance from the origin), we can't prove that two `Points` have the same corner coordinates if they are equal as determined by `equals`. That means we can't prove that this last implementation of `hashCode` is guaranteed to be correct.**

**Question 9.** (5 points) One of our major clients says that the new `Rectangle` class would be much more useful if they could access the corners of a `Rectangle`. The new intern, A. Hacker (back for yet another summer with us!), has proposed adding two new observer methods to Rectangle to make this possible:

```
/** return upper-left corner of this */
public Point getUL() { return ul; }

/** return lower-right corner of this */
public Point getLR() { return lr; }
```

Some of the other programmers, having taken CSE 331 in the past, are worried that adding these methods could have potential representation exposure problems.

Do these new methods create a rep exposure problem? Give a brief justification for your answer.

**No. `Points` are immutable. So although these methods will share references to the corner `Point` objects with clients, the clients cannot modify them.**

**Note: `Point` is specified to be an immutable type, so we are entitled to make that assumption when answering this question. A programmer could create a mutable Java subclass of `Point` and use instances of that to construct a `Rectangle`, and that would create a representation exposure problem, but it does so by using a Java type that is not a true subtype of `Point`. We might want to do some sort of defensive copying to protect against that possibility, or maybe use our `checkRep` for `Rectangle` to verify that the corner `Points` do not change after the `Rectangle` is constructed.**

**Question 10.** (10 points, 1 each) Another oldie but goodie question. Given the Java class listings on previous pages, we know that the following Java subclass relationships hold between the various point classes:

```
MutablePoint extends Point        Point3D extends Point
ColorPoint extends Point          ColorPoint3D extends ColorPoint
```

Now suppose we have the following variables:

```
Object o;
Point p;                    Point3D p3d;
ColorPoint cp;              ColorPoint3D cp3d;

List<? extends Point> lep;
List<? extends ColorPoint> lecp;
List<? super   ColorPoint> lscp;
```

For each of the following, circle OK if the statement has correct Java types and will compile without type-checking errors; circle ERROR if there is some sort of type error.

OK **ERROR** `lep.add(p);`

OK **ERROR** `lep.add(cp);`

OK **ERROR** `lecp.add(cp3d);`

**OK** ERROR `lscp.add(cp);`

**OK** ERROR `lscp.add(null);`

**OK** ERROR `o = lep.get(1);`

**OK** ERROR `p = lep.get(1);`

OK **ERROR** `cp = lscp.get(1);`

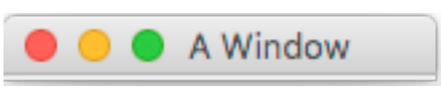
**OK** ERROR `cp = lecp.get(1);`

OK **ERROR** `cp3d = lecp.get(1);`

**Question 11.** (5 points) A little graphics debugging. One of your friends is trying to figure out Java Swing graphics, and he's come up with the following tiny program to draw an oval in a window on the screen.

```
public class Oval {
  public static void main(String[] args) {
    JFrame frame = new JFrame("A Window");
    JPanel panel = new SimplePainting();
    panel.setPreferredSize(new Dimension(300,200));
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.pack();
    frame.setVisible(true);
  }
}
class SimplePainting extends JPanel {
  @Override
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    g2.setColor(Color.yellow);
    g2.fillOval(40,30,120,100);
  }
}
```

Unfortunately when the program runs the only thing that appears is the menu title bar:




What's wrong? What needs to be done to the program so that the yellow oval is drawn properly in the window? Indicate what code needs to be added/deleted/changed and where the modification(s) need to be done. If you don't remember the exact details of some code that's needed, be as clear as you can about what is needed and we'll award partial credit accordingly.

**The panel was never added to the frame, so it is not included when the frame is drawn. The solution is to add the method call `frame.add(panel);` (or `frame.add(panel, BorderLayout.CENTER);` which does the same thing) to the code in `main` before packing the frame.**

**Question 12.** (5 points, 1 each) A last look at Campus Maps.

The HW9 Campus Paths application used the Model-View-Controller design pattern to organize the code. As with all good software, we want to make some changes to it.

For each of the following possible changes, circle Model, View, and/or Controller to indicate which component(s) of the application would need to be modified to implement that change. Each part is worth 1 point. To get the point you need to circle exactly the right answer(s).

You should answer the question based on a cleanly designed MVC organization for Campus paths, even if your own code was somewhat different.

(a) Modify the shortest-path-finding algorithm (Dijkstra's) to use integer edge weights instead of doubles.

Model View Controller

(b) Change the color of a path displayed on the map to blue.

Model View Controller

(c) Remove the reset button from the application window and automatically reset the picture whenever the user hits the delete key.

Model View Controller

(d) Replace all uses of ArrayList in the graph implementation with LinkedList.

Model View Controller

(e) Change the application so that building names are displayed in lower-case letters only.

Model View Controller

*Congratulations and have a great summer!!*
*The CSE 331 staff*