CS 3331 — Advanced Object-Oriented Programming

Answers to
Exam 1 on UML, Java, Unit Testing, and Inheritance

This test has 3 questions and pages numbered 1 through 9.

Reminders

This test is open book and notes; but no laptop computers, PDAs, calculators, or similar devices are allowed. However, it is to be done individually, and you are not to exchange or share materials with other students during the test.

If you need more space, use the back of a page, noting this on the front.

This test is timed. Your test will not be graded if you try to take more than the time allowed. Therefore, before you begin, please take a moment to look over the entire test so that you can budget your time.

For programs and diagrams, clarity is important; if your programs or diagrams are sloppy and hard to read, you will lose points. Correct syntax also makes some difference.

There are 100 points all.

1. (10 points/6 mins) Briefly answer the following questions:

(a) Explain the life-cycle of applets by describing when applet methods such as init, start, stop, and destroy are invoked by a web browser or an applet viewer.

Answer: The init method is invoked when the applet class is initially loaded for execution, the start and stop methods are invoked when entering and leaving the web page that contains the applet, and the destroy method is invoked when the web page is discarded.

(b) What does the substitution property mean in object-oriented programming languages such as Java? Give an example of it.

Answer: It means that an object of a subtype can be used in a place where an object of a supertype is required, e.g., as arguments, return values, receivers, and RHS of assignments. For example, aStack.push(new Integer(10)) works if the push method’s parameter type is Object.
2. (25 points/10 mins) Write a JUnit test class named ModuloCounterTest for the class ModuloCounter by filling out the skeleton code given below. Your test class should include test methods for all methods and constructors of the class ModuloCounter. If the method or constructor under test can terminate abruptly by throwing an exception, you should also include test data for such cases.

```java
public class ModuloCounter {

    private final int mod;
    private int val = 0;

    public ModuloCounter(int mod) {
        if (mod > 0) {
            this.mod = mod;
        } else {
            throw new IllegalArgumentException("Invalid argument " + mod);
        }
    }

    public int mod() {
        return mod;
    }

    public int val() {
        return val;
    }

    public void incr() {
        val = (val + 1) % mod;
    }
}

/** A JUnit test class to test the class {link ModuloCounter}. */
public class ModuloCounterTest extends TestCase {

    /** Creates a new instance. */
    public ModuloCounterTest(String name) {
        super(name);
    }

    /** Returns the test suite for this test class. */
    public static Test suite() {
        return new TestSuite(ModuloCounterTest.class);
    }

    /** Runs the tests. */
    public static void main(String[] args) {
        junit.textui.TestRunner.run(suite());
    }

    // WRITE YOUR TEST METHODS HERE (MORE SPACE ON THE NEXT PAGE)...

    Answer:
```
/** Tests the constructor. */
public void testModuloCounter() {
    // normal case
    ModuloCounter ctr = new ModuloCounter(1);
    assertEquals(1, ctr.mod());
    assertEquals(0, ctr.val());

    // exceptional case
    try {
        ctr = new ModuloCounter(0);
        fail();
    } catch (IllegalArgumentException e) {
    } catch (Exception e) {
        fail();
    }
}

/** Tests the mod method. */
public void testMod() {
    assertEquals(10, new ModuloCounter(10).mod());
}

/** Tests the val method. */
public void testVal() {
    ModuloCounter ctr = new ModuloCounter(10);
    assertEquals(0, ctr.val());
    ctr.incr();
    assertEquals(1, ctr.val());
}

/** Tests the incr method. */
public void testIncr() {
    final int m = 5;
    ModuloCounter ctr = new ModuloCounter(m);
    for (int i = 0; i < 3 * m; i++) {
        ctr.incr();
        assertEquals((i + 1) % m, ctr.val());
    }
}
} // End of ModuloCounterTest
3. (65 points total/50 mins) This problem consists of four subproblems and is about writing an animation applet that bounces different kinds of balls on the screen. You are to write the main applet class and several helper classes whose design and partial implementation are given below. The applet bounces two balls in the screen. A ball changes (i.e., reverses) its direction if it touches any of the four sides of the screen. One ball is a (filled) circle and the other is a text. The applet uses the animation idiom and the double buffering technique. In subproblems (a)–(d) below, you are to fill in the missing design and implementation of the classes, some of whose skeleton code is provided. The whole program consists of one interface (Bounceable) and four classes (BouncingApplet, BouncingObject, BouncingCircle, and BouncingText). Before you work on individual subproblems, it might be a good idea to look at other subproblems, for the subproblems are related to each others.

(a) (10 points/10 mins) Complete the design of the bouncing applet program by adding to the following UML class diagram various relationships among the classes and the interface. For associations, aggregates, and compositions, include direction and multiplicity information. For the interface Bounceable and the class BouncingObject, specify field and method declarations; for this, use the standard UML notation. (Hint: refer to subproblems (b)–(d) below for the definitions of BouncingApplet, Bounceable, and BouncingObject.)
(b) (10 points/5 mins) Given the interface Bounceable below, complete the class BouncingApplet by filling in the definition of the paintFrame method on page 6. The import statements are suppressed to save space.

```java
/** An interface denoting bounceable objects. An object can be
 * bounced if it has the <code>bounce</code> method. */
public interface Bounceable {
    /** Bounces this object inside the screen of dimension
     * <code>d</code> by using the given graphics <code>g</code>. This
     * method will be called continuously and is supposed to adjust
     * the current position of the object and paint it. */
    void bounce(Graphics g, Dimension d);
}
```

```java
/** An animation applet to show several kinds of moving objects
 * bouncing around the screen. */
public class BouncingApplet extends Applet implements Runnable {
    /** The dimension of this applet’s screen. */
    private Dimension dim;

    /** The set of balls to bounce. */
    private Bounceable[] balls;

    /** Initializes the applet by creating several bounceable objects. */
    public void init() {
        dim = getSize();
        balls = new Bounceable[] {
            // circle with the initial position, color and radius.
            new BouncingCircle(50, 50, Color.RED, 10),
            // text with the initial position, color and string
            new BouncingText(10, 30, Color.YELLOW, "I Love Exam!"),
        };
    }

    // ---------------------------------------------
    // DOUBLE_BUFFERING
    // ---------------------------------------------
    /** The off-screen image for double buffering. */
    protected Image image;

    /** The off-screen graphics for double buffering. */
    protected Graphics offScreen;

    /** Updates this applet on the graphics <code>g</code> by calling
     * the hook method <code>paintFrame</code>. */
    public void update(Graphics g) {
        if (image == null) {
            image = createImage(dim.width, dim.height);
            offScreen = image.getGraphics();
        }
        paintFrame(offScreen);
        g.drawImage(image, 0, 0, this);
    }
```
/** Bounces each ball of the array <code>balls</code> by calling its <code>bounce</code> method. */
protected void paintFrame(Graphics g) {
  // WRITE YOUR CODE HERE!
  g.setColor(Color.BLACK);
  g.fillRect(0, 0, dim.width, dim.height);
  for (int i = 0; i < balls.length; i++) {
    balls[i].bounce(g, dim);
  }
}

/** Rewires to <code>update</code> method to use double buffering. */
public void paint(Graphics g) {
  update(g);
}

// ANIMATION IDIOM
// ----------------------------------------------------------------------
/** The animation thread. */
protected Thread animationThread;

/** The refresh rate in milliseconds. */
protected int delay = 10;

/** Starts a new animation thread, if not yet started. */
public void start() {
  if (animationThread == null) {
    animationThread = new Thread(this);
    animationThread.start();
  }
}

/** Stops the animation thread. */
public void stop() {
  animationThread = null;
}

/** Calls the repaint method once in every <code>delay</code> milliseconds. */
public void run() {
  while (Thread.currentThread() == animationThread) {
    repaint();
    try {
      Thread.currentThread().sleep(delay);
    } catch (InterruptedException e) {} 
  }
}

(c) (20 points/15 mins) Given the abstract class `BouncingObject` below, write the concrete class `BouncingCircle` (Hint: refer to the design given in subproblem (a) and a sample use by the class `BouncingApplet` in subproblem (b). The detailed design of this class is found on page 4; i.e., you only need to add one field, one constructor and one method.)

```java
/**
 * An abstract class representing various bouncing objects. A
 * bouncing object has the current position, color, and moving speed.
 */
public abstract class BouncingObject implements Bounceable {

    /** The current position of this object. */
    protected int x, y;

    /** The color of this object. */
    protected Color color;

    /** The x and y speeds of this object; i.e., the object moves dx and
        * dy pixels each time the <code>bounce</code> method is called. */
    protected int dx, dy;

    /** Creates a new instance with the given initial position and
        * and color. The dx and dy are set to default values. */
    protected BouncingObject(int x, int y, Color color) {
        this.x = x;
        this.y = y;
        this.color = color;
        dx = 2;
        dy = 2;
    }

    // WRITE YOUR ANSWER ON THE NEXT PAGE BY FILLING IN THE GIVEN TEMPLATE!
```
/** A concrete class representing bouncing circles. */
public class BouncingCircle /* YOUR CODE HERE -> */ {

extends BouncingObject

/** The radius of this circle. */
private int radius;

/** Creates a new circle of given position, color, and radius. */
public BouncingCircle(int x, int y, Color color, int radius) {
    super(x, y, color);
    this.radius = radius;
}

/**
 * Bounces this circle in the screen of dimension <code>d</code>. The position of the circle is adjusted based on the x- and y-speeds (dx and dy) and the circle is painted at the new position. */
public void bounce(Graphics g, Dimension d) {
    // reverse the direction if touches any of the four sides.
    if (x < radius || x + radius > d.width) {
        dx = -dx;
    }
    if (y < radius || y + radius > d.height) {
        dy = -dy;
    }

    // move the circle
    x += dx;
    y += dy;

    // draw the circle
    Color oldColor = g.getColor();
    g.setColor(color);
    g.fillOval(x - radius, y - radius, 2*radius, 2*radius);
    g.setColor(oldColor);
}
} // End of BouncingCircle
(d) (25 points/20 mins) Write the concrete class `BouncingText`. (Hints: refer to subproblems (a) – (c) and a sample use by `BouncingApplet` in subproblem (b); use the `FontMetrics` class (on page 193 of textbook) to measure the size of a text. The detailed design of this class is found on page 4; i.e., you only need to add one field, one constructor and one method.)

```java
/** A concrete class representing bouncing texts. */
public class BouncingText /* YOUR CODE HERE -> */ {

    /** The font to paint this bouncing text. */
    private Font font = new java.awt.Font("Sans serif", Font.BOLD, 24);

    /** The text to bounce. */
    private String text;

    /** Creates a new bouncing text of given initial position, color, *
     * and text. */
    public BouncingText(int x, int y, Color color, String text) {
        super(x, y, color);
        this.text = text;
    }

    /** Bounces this text in the screen of dimension <code>d</code>. *
     * The position of the circle is adjusted based on the x- and *
     * y-speeds (dx and dy) and the text is painted at the new position. */
    public void bounce(Graphics g, Dimension d) {
        // get the size measurement
        Font oldFont = g.getFont();
        g.setFont(font);
        FontMetrics fm = g.getFontMetrics();
        int length = fm.stringWidth(text);

        // reverse the direction if touches any of the four sides.
        if (x < 0 || x + length > d.width) {
            dx = -dx;
        }
        if (y - fm.getAscent() < 0 || y + fm.getDescent() > d.height) {
            dy = -dy;
        }

        // change the position
        x += dx;
        y += dy;

        // draw the text
        Color oldColor = g.getColor();
        g.setColor(color);
        g.drawString(text, x, y);
        g.setColor(oldColor);
        g.setFont(oldFont);
    }

    } // End of BouncingText
```