Reminders

This test is closed-notes and closed-book. However, you are allowed to bring 1 page (8.5 X 11) of notes (both sides). Your notes must be your own, they must be hand written, and they must be turned in with your test. This test is to be done individually, and you are not to exchange or share your notes with other students during the test.

If you need more space, use the back of a page. Note when you do that 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 diagrams and programs, clarity is important; if your diagrams or programs are sloppy or hard to read, you will lose points. Correct syntax also makes some difference.

There are 110 points all, including 10 bonus points.

1. (total 20 points) Short answers: multiple choice and fill-in-the-blank

   (a) (2 points) Software development is difficult, time-consuming, and costly. Which of the followings is less likely to be one of the main causes of the problems in software development?
      
      i. Complexity
      ii. Longevity and evolution
      iii. High user expectation
      iv. New programming languages

   (b) (2 points) There are many desirable qualities of software systems, including usefulness, timeliness, reliability, maintainability, reusability, user friendliness, and efficiency. The object-oriented development approach cannot directly improve all the desirable qualities of software systems. It focuses primarily on improving the ______________________ and ______________________ of software systems.

   (c) (2 points) The SOLID principles consist of five principles and are well-known object-oriented design principles to produce more understandable, flexible and maintainable designs. Name one design principle of SOLID (i.e., S, O, L, I, or D): ______________________

   (d) (2 points) A UML ______________________ diagram is for modeling the requirements of a software system. It provides an important link between the requirements and an object-oriented analysis and design model.

   (e) (2 points) An object-oriented framework is a semi-complete application, providing the basic structure (or backbone) and utilities for applications to allow one to build an application quickly. And a(n) ______________________ refers to the fact that the top-level control of an application resides in a framework itself, not in the code written by an application developer.
(f) (2 points) A UML class diagram can express several kinds of relationships between classes. Which of the following relationships is the strongest (i.e., most tightly coupled)?

i. Association
ii. Aggregation
iii. Composition
iv. Dependency

(g) (2 points) Which of the followings is an incorrect statement about a UML state machine diagram?

i. It describes the static structure of a system.
ii. It depicts the flow of control using states and transitions.
iii. It is a generalization of the finite state machine.
iv. It is well suited for modeling a reactive system.

(h) (2 points) All of the following tags are used to specify the interface (input and output) of a method or constructor, EXCEPT for:

i. @param
ii. @return
iii. @throws
iv. @see

(i) (2 points) There are two kinds of exceptions: checked exception and unchecked exception. Is the NullPointerException class a checked or unchecked exception?

i. Checked
ii. Unchecked

(j) (2 points) A(n) ____________________ interface is an interface with exactly one non-default method. A lambda expression may be used in any place where an object of this interface is required, including method arguments, method return value, and assignments. Examples of this interface include ActionListener, Consumer<T>, and BoardPanel.ClickListener.

2. (5 points) When is a type (a class, an interface, an array, and a generic class) a subtype of another type in Java?

3. (5 points) Name two forms of polymorphisms supported in Java, and compare them briefly.
4. (15 points) Draw a UML class diagram to show the detailed design of the following Java skeleton code. In the code, a custom @NonNull annotation is used to assert that the annotated type/field can't be null. Your diagram should show all the design/implementation decisions of the code, including field/method declarations.

```java
interface A {} // Q: interface?

interface B extends A {} // Q: extends?

abstract class C implements A {} // Q: abstract, implements?

class D {
    private Set<B> bs; // Q: object structure?
    private @NonNull E e; // @NonNull: e can't be null.
    private int f; // Q: field declaration?
    public int m1() {} // Q: method declarations?
    protected void m2(int x) {} // Q: method declarations?
    void m3() {} // Q: method declarations?
}

class E extends C { // Q: extends?
    private @NonNull D d; // Q: object structure?
    private E e;
}
```
5. (total 20 points) Consider the following partial code where a board consists of \( n \times n \) squares. In the sub-questions below, you are to write the `equals` and other methods for the `Square` class.

```java
public class Board {
    private final List<Square> places;

    public Board(int size) {
        places = new ArrayList(size * size);
        for (int x = 0; x < size; x++) {
            for (int y = 0; y < size; y++) {
                places.add(new Square(this, x, y));
            }
        }
    }
    ...
}

public class Square {
    public final Board board;
    public final int x, int y;
    private int number;

    public Square(Board board, int x, int y) {
        this.board = board; this.x = x; this.y = y;
    }
    ...
}

(a) (5 points) Let’s first rewrite the constructor of the `Board` class to make it more robust by throwing an `InvalidSizeException` when the given size is not a square number (or perfect square). Assume that:

- The `InvalidSizeException` class is already defined as a direct subclass of the `Exception` class.
- There is a helper method defined, `boolean isPerfectSquare(int)`, that checks if an integer is a square number.
(b) (5 points) State the contract of the `equals` method inherited from the class `Object`, i.e., properties or constraints that its implementation has to satisfy.

(c) (6 points) Write the `equals` method for the `Square` class. Your method should override the one inherited from the `Object` class. Assume two places are equal if they have the same coordinates (x and y values).

(d) (4 points) The above code of the `Square` class has one serious problem in that there is a missing method that needs to be overridden. Name the method and provide its definition (code).
6. (5 points) Write a JUnit test method equivalent to the `testBoard` method shown below, but without using the optional parameter `expected` of the `@Test` annotation.

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class BoardTest {

    @Test(expected=InvalidSizeException.class)
    public void testBoard() {
        new Board(7);
    }
    ...
}
```
7. (total 15 points) Consider the following applet that animates a bouncing ball (a filled circle), initially positioned at the center of the screen.

```java
public class BallApplet extends Applet {

    protected Dimension dim; // screen dimension
    private Timer timer; // animation timer

    // attributes of the bouncing ball
    private int x, y; // current position
    private int dx = 2, dy = 4; // bouncing direction and distance
    private int radius = 20; // size
    private Color color = Color.GREEN; // color

    /** Initialize the ball and the animation timer. */
    public void init() {
        dim = getSize();
        x = dim.width / 2;
        y = dim.height / 2;
        timer = new Timer(100, new ActionListener() { // <-- Question (a)
            public void actionPerformed(ActionEvent event) {
                repaint();
            }
        });
    }

    /** Bounce the ball. */
    public void paint(Graphics g) {
        g.setColor(Color.BLACK);
        g.fillRect(0, 0, dim.width, dim.height);
        if (x < radius || x > dim.width - radius) {
            dx = -dx;
        }
        if (y < radius || y > dim.height - radius) {
            dy = -dy;
        }
        x += dx;
        y += dy;
        g.setColor(color);
        g.fillOval(x - radius, y - radius, radius * 2, radius * 2);
    }

    public void start() { timer.start(); }
    public void stop() { timer.stop(); }
}

(a) (5 points) The timer field is initialized in the init method. Rewrite the timer initialization statement using the Java 8 lambda notation.
(b) (10 points) Draw a UML behavior state machine diagram to model the bouncing ball. Your model should include at least four state variables appearing in the code: \( x, y, dx, \) and \( dy \); describe changes of their values. You may use the following abbreviations to denote different events \( (e) \) and conditions \( (c) \):

- \( e_s \): animation started or resumed
- \( e_p \): animation stopped (paused to be more precise)
- \( e_t \): 100 milliseconds elapsed
- \( c_x \): a side wall is touched by the ball
- \( c_y \): a top/bottom wall is touched by the ball
8. (total 15 points) In this question, you are to improve the structure of the bouncing ball applet by separating the bouncing ball from the applet. You will reuse the `BallApplet` class from the previous question by introducing a new subclass named `ModularBallApplet`. An improved design is shown below along with the source code of the new applet. A bouncing ball applet is now composed of a ball existing as a separate object, and it communicates with the ball through a well-defined interface, `Ball`.

```
public class ModularBallApplet extends BallApplet {
    /** Ball to bounce. */
    private Ball ball;

    /** Overridden here to create a bouncing ball. */
    public void init() {
        super.init();
        ball = createBall();
    }

    /** Create a bouncing ball. */
    protected Ball createBall() {
        return new BouncingBall(dim);
    }

    /** Overridden here to draw the ball. */
    public void paint/Graphics g) {
        g.setColor(Color.BLACK);
        g.fillRect(0, 0, dim.width, dim.height);
        ball.draw(g);
    }
}
```

(a) (5 points) Write the interface `Ball` (Hint: how is the interface used by the `ModularBallApplet` class?)
(b) (10 points) Write the class BouncingBall by completing the following skeletal code.

```java
public class BouncingBall implements Ball {
    private int x, y;
    private int dx = 2, dy = 4;
    private int radius = 20;
    private Color color = Color.GREEN;
    private Dimension dim; // screen dimension

    // WRITE YOUR CODE HERE ...
}
```
9. (10 points) Write a subclass of the `ModularBallApplet` class from the previous question, named `OrbitingBallApplet`, to show a ball that circles around at a fixed distance from the center of the screen (see a sample screenshot below). Introduce a new ball class, say `OrbitingBall` to represent such a ball. As suggested below, it would be a good idea to use the polar notation (distance and angle) to denote the current position of the ball: use the provided helper methods to convert a polar value to x- and y-coordinates. Hint: The `OrbitingBallApplet` class needs to override only one simple method.

```java
public class OrbitingBallApplet extends ModularBallApplet {
    // WRITE YOUR CODE HERE (A) ...
}

public class OrbitingBall ...
    // WRITE YOUR CODE HERE (B) ...

/** Calculate the x-coordinate of the current position. */
private int calX() {
    int center = dim.width / 2;
    return (int) (center + distance * Math.cos(Math.toRadians(angle))); // angle in degrees (0-360)
}

/** Calculate the y-coordinate of the current position. */
private int calY() {
    int center = dim.height / 2;
    return (int) (center + distance * Math.sin(Math.toRadians(angle)));
}
```