Your name: _____

CSSE 221 – Introduction to Software Development Honors

Exam 1

Closed-book problems: You can use a single 8.5 inch by 11 inch "cheat sheet" (back and front, with whatever you want on it). Nothing else. When you are done with this section, turn it in and get the rest of the test.

Each of the closed-book problems are 1 point except as noted. There is a total of 40 points in this section.

1. What keyword indicates that a ***method does not return a value***? void

2. What keyword indicates that ***a variable cannot change its value from its initial value***? final

3. (2 points) Give four examples of ***primitive types***.
   int long byte short char double float boolean

4. (2 points) What is the ***implicit parameter*** in the following? The ***explicit parameters***?
   ```
   father.replace('a', '8');
   ```
   father is the implicit parameter, *'a'* and *'8'* are the explicit parameters

5. (2 points) In a class definition, what keyword refers to the ***implicit parameter***? this

6. (2 points) Explain what space is allocated by the following:
   ```
   Rectangle box = new Rectangle(5, 10, 20, 30);
   ```
   Space for the four numbers associated with the Rectangle object (x coordinate, y coordinate, width, height), plus space for the box variable

7. Generally, what ***visibility*** should ***fields*** have? private

8. What ***visibility*** should a ***method*** have if you want it visible to objects of ANY class? public

9. (2 points) If class B ***implements interface X***, what does that imply? The class B must contain implementations of the methods specified by the interface X.

10. (2 points) If class B ***extends class Y***, what does that imply?
    The class B inherits the fields and methods implemented in class Y.

11. (5 points) Consider the Eye class, part of which is shown here:

```
public class Eye extends JPanel implements MouseMotionListener {
   protected EyeBall eyeBall;
   protected Color eyeColor;
   protected int eyeRadius;

   public Eye() {
         this(JavaEyes.DEFAULT_EYE_COLOR, JavaEyes.DEFAULT_EYEBALL_COLOR);
   }

   public Eye(Color eyeColor, Color eyeBallColor) {
         this.eyeColor = eyeColor;
         ...
   }


   public void mouseMoved(MouseEvent event) {
         Point mousePoint = new Point(event.getX(), event.getY());
         this.look(mousePoint);
   }

   protected void look(Point mousePoint) {
         int eyeX = this.getX() + this.eyeRadius; // Center of eye,
         int eyeY = this.getY() + this.eyeRadius; // relative to the frame
               ...
   }

   @Override
   public void paintComponent(Graphics graphics) {
         super.paintComponent(graphics);
         ...
   }
}
```

a. How many fields are there, and what are their names? 3: eyeBall, eyeColor, eyeRadius

b. How many constructors are there? Two

c. How many methods are there and what are their names? 3: mouseMoved, look, paintComponent.

d. What is the name of one of the parameters (any one you pick is fine)? event, mousePoint, graphics

e. What is the name of one of the local variables that is not a parameter (any one you pick is fine)? mousePoint, eyeX, eyeY

12. (2 points) On the previous page, the *eyeColor* variable is NOT *static*. Suppose that we made it *static* – what would be the effect of that?

    The variable would be shared by all instances of the Eye class, so all Eye's would have the same eyeColor.

13. What does ***foo overrides blah*** mean? NOTE: this question is a bit confusing, since *foo* and *blah* would have to be the same name for the question to make sense.

    That said, here is an answer: methods *foo* and *blah* have the same signature (hence the same name), with *foo* appearing in a subclass of the class containing method *blah*. Objects in the subclass will run *foo* instead of (or possibly in addition to, if *blah* contains a super.foo() call) *blah*.

14. What does it mean for a method to be ***overloaded***? The method has two definitions with the same name but different signatures.

15. What class is at the top of the ***inheritance hierarchy***? Object

16. Does Java allow ***multiple inheritance***? No

17. (2 points) Every object has a ***declared type***, and every non-null object has an ***actual type***. Give an example that shows the difference.

    List<Cat> cats = new ArrayList<Cat>();

    The left-hand-side is the declared type (List<Cat>); the right-hand-side is the actual type (ArrayList<Cat>).

18. (2 points) What is the ***notation for doing a cast*** of the expression *x * 6.4* to type *int*? (int) (x * 6.4)

19. (2 points) Give two reasons why ***UML class diagrams*** are useful.

    1. To develop a design of the classes and their relationships.
    2. To communicate a design of the classes and their relationships.

20. What is the UML notation for ***is-a (extends)***?  Actually, the arrow head should be an open triangle, not a filled one, in this and the next problem.

21. What is the UML notation for ***is-a (implements)***? 

22. What is the UML notation for ***has-a***? 

23. What is ***cohesion*** in class design? A class is cohesive if it is a single concept.

24. Do you want ***high cohesion*** or ***low cohesion*** in the classes that you design? (Circle your choice.)

25. What is ***coupling*** in class design? Coupling refers to the number of classes that the given class depends on.

26. Do you want ***high coupling*** or ***low coupling*** in the classes that you design? (Circle your choice.)

Your name: _____

Open-book problems: You may use any non-human source, e.g. your computer, the Internet, your text, the course web site, and your notes.

1. (10 points) Write statements that implement the following method per its description, using an explicit loop (no fair using the *Arrays.max* method):

```
// Returns the largest number in the given array.
// Assume that the array contains at least one number.
private Double largest(Double[] numbers) {
    double max = numbers[0];        // A Double ('D' not 'd') works too.
    for (Double number : numbers) {
        if (number > max) {
            max = number;
        }
    }
    return max;
```

Or:

```
    double max = numbers[0];
    for (int k = 1; k < numbers.length; ++k) {
        if (number[k] > max) {
            max = number[k];
        }
    }
    return max;
```

2. (10 points) Write statements that construct an ArrayList of String's and initializes the ArrayList to contain 100 String's, each of which is "Hello".

```
List<String> list = new ArrayList<String>();
for (int k = 0; k < 100; ++k) {
    list.add("Hello");
}
```

On-the-computer problem

1. (40 points) Go to the SVN Repository Perspective and Checkout your ***Exam1*** project from your individual repository.

   The project has the *StringTransformable* interface that you used in WordGames.

   Use the following steps to implement the ***Quitter*** class described below, much as you did for the WordGames project:

   - Step 1: Create the class and add **documented stubs** for the constructors and methods of the class.
   - Step 2: Write **JUnit 4 tests** for the *transform* method of the class.
     - Thorough unit-testing might require a dozen or so tests per class, but in order to keep this project from being too much work, you should have **TWO well-chosen tests for this class**.
     - **Your Quitter code must have good style and documentation**, but your JUnit test class need NOT have any documentation and will NOT be graded on style. That is, I'll grade the quality of your two tests, but nothing else in your JUnit test class.
   - Step 3: Implement the class, as follows:
     a. Implement the code for the class.
     b. Run your JUnit tests for the class.
     c. Correct your code and/or unit tests as needed, and modify your documented stubs as needed.
   - Step 4: When your class is correct, commit your project.
     - Make sure that your commit has the *src* folder checked.
     - As always, include an appropriate comment, e.g. "Quitter finished" when you finish the *Quitter* class.

   A Quitter has a positive whole number (I'll call it *n*). For the first *n* String's that a Quitter is given, it returns its given String with " **- I am on it!**" appended. (That is a single space, a hyphen, another space, the words *I am on it* and an exclamation point.) After that, the Quitter simply returns its given String unchanged.

   If the Quitter is not told its number, it should use 1 as its number (so that it quits appending " - I am on it!" after having done the appending only once).

   For example, one Quitter might have 3 as its number. Given the following String's (in the order listed), it acts as indicated:
   - Given "Hello", returns "Hello - I am on it!"
   - Given "Hi there", returns "Hi there - I am on it!"
   - Given "Oops!", returns "Oops! - I am on it!"
   - Given "OK, stop fooling around", returns "OK, stop fooling around"
   - Given "Hello", returns "Hello"

   Just ask if you are not sure what a Quitter does. Grading rubric:

   - Correctness: 20 points (10 points for right constructor(s), 10 points for *transform* is correct)
   - Good JUnit tests: 10 points
   - Uses good style (Quitter class only, style is not graded in your JUnit class): 5 points
   - Has appropriate documentation (Quitter class only, no need for any documentation in your JUnit class): 5 points

```java
/**
 * A Quitter keeps track of the number of times that this Quitter has been asked
 * to transform, and if that number is less than or equal to this Quitter's
 * quitting time, then the returned String is the given String with
 * "- I am on it" appended. After the quitting time, the returned String is the
 * given String, unchanged.
 * <p>
 * Each Quitter has its own quitting time; the default quitting time is 1.
 *
 * @author David Mutchler. Created October 8, 2009.
 */
public class Quitter implements StringTransformable {
	private static String messageToApppend = " - I am on it!";
	private int quittingTime;
	private int numberOfTransformationsSoFar;

	/**
	 * Initializes the quitting time to 1 (so that the first transform returns
	 * the given String with the special message appended while subsequent
	 * transforms return the given String unchanged.
	 */
	public Quitter() {
		this(1);
	}

	/**
	 * Initializes the quitting time to the given value (so that transforms
	 * after the quitting time return the String unchanged, while transforms at
	 * or before the quitting time return the String with a message appended.
	 *
	 * @param quittingTime
	 *            Number of times to append the string before quitting.
	 */
	public Quitter(int quittingTime) {
		this.quittingTime = quittingTime;
		this.numberOfTransformationsSoFar = 0;
	}

	/**
	 * Keeps track of the number of times that this Quitter has been asked to
	 * transform, and if that number is less than or equal to this Quitter's
	 * quitting time, then the returned String is the given String with
	 * "- I am on it" appended. After the quitting time, the returned String is
	 * the given String, unchanged.
	 *
	 * @param stringToTransform
	 *            The String to transform.
	 * @return The transformed String.
	 */
	@Override
	public String transform(String stringToTransform) {
		++this.numberOfTransformationsSoFar;
		if (this.numberOfTransformationsSoFar <= this.quittingTime) {
			return stringToTransform + Quitter.messageToApppend;
		} else {
			return stringToTransform;
		}
	}
}
```

```java
import static org.junit.Assert.assertEquals;

import org.junit.Test;

/**
 * A QuitterTest tests the Quitter class. In particular, it tests the Quitter's
 * transform method.
 *
 * @author David Mutchler. Created October 8, 2009.
 */
public class QuitterTest {
	/**
	 * Test method for {@link Quitter#transform(java.lang.String)}.
	 */
	@Test
	public void testQuitAfter5() {
		Quitter quitAfter5 = new Quitter(5);

		assertEquals("First string - I am on it!", quitAfter5
				.transform("First string"));

		assertEquals(" - I am on it!", quitAfter5.transform(""));

		assertEquals("Third string - I am on it!", quitAfter5
				.transform("Third string"));

		assertEquals("Fourth string, testing a longer one. - I am on it!",
				quitAfter5.transform("Fourth string, testing a longer one."));

		assertEquals("Fifth string - I am on it!", quitAfter5
				.transform("Fifth string"));

		// Should quit appending from here on.
		assertEquals("Sixth string", quitAfter5.transform("Sixth string"));

		assertEquals("Seventh string-!", quitAfter5
				.transform("Seventh string-!"));

		assertEquals("Eighthhhh", quitAfter5.transform("Eighthhhh"));

		assertEquals("Ninth string", quitAfter5.transform("Ninth string"));

		assertEquals("TENTH", quitAfter5.transform("TENTH"));

		assertEquals("11", quitAfter5.transform("11"));
	}

	/**
	 * Test method for {@link Quitter#transform(java.lang.String)}.
	 */
	@Test
	public void testDefaultQuitter() {
		Quitter quitAfter1 = new Quitter();
		Quitter quitAfter1Too = new Quitter();

		// quitAfter1Too starts out appending.
		assertEquals("First string - I am on it!", quitAfter1
				.transform("First string"));

		// quitAfter1 quits appending from here on.
		assertEquals("", quitAfter1.transform(""));
		assertEquals("Third string", quitAfter1.transform("Third string"));
```

```
        // quitAfter1Too starts out appending.
        assertEquals(" - I am on it!", quitAfter1Too.transform(""));

        // quitAfter1Too quits appending from here on.
        assertEquals("", quitAfter1Too.transform(""));
        assertEquals("Third string", quitAfter1Too.transform("Third string"));
    }

    /**
     * Test method for {@link Quitter#transform(java.lang.String)}.
     */
    @Test
    public void testQuitterInteractions() {
        Quitter quitAfter5 = new Quitter(5);
        Quitter quitAfter1 = new Quitter();
        Quitter quitAfter1Too = new Quitter();

        // QuitAfter1 appends on its first transform.
        assertEquals("First string - I am on it!", quitAfter1
                    .transform("First string"));

        // QuitAfter1Too appends on its first transform.
        assertEquals(" - I am on it!", quitAfter1Too.transform(""));

        // QuitAfter1 stops appending after its first transform.
        assertEquals("ABC", quitAfter1.transform("ABC"));
        assertEquals("DEF", quitAfter1.transform("DEF"));

        // QuitAfter5 appends on its first transform.
        assertEquals("First string - I am on it!", quitAfter5
                    .transform("First string"));

        // QuitAfter1Too stops appending after its first transform.
        assertEquals("", quitAfter1Too.transform(""));

        // QuitAfter1 is finished appending.
        assertEquals("OK OK OK ", quitAfter1.transform("OK OK OK "));

        // QuitAfter1Too is finished appending.
        assertEquals("", quitAfter1Too.transform(""));
        assertEquals("", quitAfter1Too.transform(""));
        assertEquals("Third string", quitAfter1Too.transform("Third string"));
        assertEquals("Third string", quitAfter1Too.transform("Third string"));

        // QuitAfter1 is still finished appending.
        assertEquals("", quitAfter1.transform(""));

        // QuitAfter5 is still appending.
        assertEquals(" - I am on it!", quitAfter5.transform(""));
        assertEquals("Third string - I am on it!", quitAfter5
                    .transform("Third string"));
        assertEquals("Fourth string, testing a longer one. - I am on it!",
                    quitAfter5.transform("Fourth string, testing a longer one."));

        // QuitAfter1 is still finished appending.
        assertEquals("Third string", quitAfter1.transform("Third string"));

        // QuitAfter5 is still appending.
        assertEquals("Fifth string - I am on it!", quitAfter5
                    .transform("Fifth string"));
```

```
            // QuitAfter5 has finally finished appending.
            assertEquals("Sixth string", quitAfter5.transform("Sixth string"));
            assertEquals("Seventh string-!", quitAfter5
                        .transform("Seventh string-!"));
            assertEquals("Eighthhhh", quitAfter5.transform("Eighthhhh"));
            assertEquals("Ninth string", quitAfter5.transform("Ninth string"));
            assertEquals("TENTH", quitAfter5.transform("TENTH"));
            assertEquals("11", quitAfter5.transform("11"));

            // QuitAfter1 is still finished appending.
            assertEquals("The end!", quitAfter1.transform("The end!"));
        }
}
```