UNIVERSITY OF WISCONSIN-LA CROSSE
Department of Computer Science

CS 120 | **Software Design I** | **Fall 2016**
Final Exam (Practice) | **21 December 2016**

- **Do not turn the page until instructed to do so.**
- **This booklet contains 12 pages including the cover page.**
- **This is a closed-book exam. All you need is the exam and a writing utensil.**
- **You have exactly 120 minutes.**
- **The maximum possible score is 80 points.**

PROBLEM	SCORE
1	10
2	10
3	10
4	5
5	10
6	10
7	10
8	15
TOTAL	80

ANSWER KEY

1. **(10 pts.) TRUE/FALSE.**

   For each of the following, indicate whether the statement is true or false.
   You do not need to explain your answers (although I sometimes do so here).

   a. Anywhere an object of class `C` can be used, a call to a non-`void` method that returns something of type `C` can be used.

      True

   b. Any variable that is *not* a global instance variable, is always local to the scope in which it is instantiated.

      False

   c. Any variable that is *not* a global instance variable is always local to the scope in which it is declared.

      True

   d. If we nest one loop inside the other, then both loops must be of the same type (i.e., they must both be `for`-loops or both `while`-loops).

      False

   e. Every result that can be achieved using a `while` loop can be achieved using a `for` loop.

      True

   f. When we override a method, the new version must have the same output `return` type as the original.

      True: this is necessary so that the returned values can be used in the same contexts.

   g. When we override a method, the new version must have input parameters with the same *variable names* as the inputs to the original.

      False: variable names don't matter (as long as they're legal).

   h. When we override a method, the new version must have input parameters with the same *types* as the inputs to the original.

      True: this is necessary so that the method calls will truly over-ride the originals.

   i. Every object conforms to the type of its descendant sub-classes.

      False: it is the other way round (objects conform to their *ancestors*).

   j. A variable defined with a given type can be used to name any object of a type that conforms to the original.

      True: this is one of the basic features of conformance.

2. **(10 pts.) SHORT ANSWER.**

a. (3 pts.) In Java, expressions are *converted automatically* from a **less precise (or *narrower*)** type to a **more precise (or *wider*)** type, but not vice-versa. For example, an expression that evaluates to `int` type **will** convert automatically to `double` type.

b. (3 pts.) The special Java keyword `this`, when used within the code of a class `C`, always refers to **the current instance of the class (the one whose method or variable is being used)**. The special keyword `super`, when used in a class `C`, always refers to **the parent class instance of the current object**. In particular, the method call `super()` always refers to **the parent class constructor**.

c. (4 pts.) Suppose we have three classes, `Class1`, `Class2`, and `Class3`. Furthermore, suppose that `Class2 extends Class1` and `Class3 extends Class2`. Assuming that each class has a no-argument default constructor, some of the following lines will compile correctly, and some will not. **Circle those lines that are correct.**

```
Class1 ca = new Class1();
Class1 cb = new Class2();
Class1 cc = new Class3();
Class2 cd = new Class1();
Class2 ce = new Class3();
Class3 cf = new Class2();
```

Explanation: all of the legal lines are example of correct conformance, where we start with a variable declaration of one type, and then assign an object of a conformant (identical or descendant) type. The ones that are not legal are those where there is no conformance. (The rules here are the same as when we use an object in a method call.)

3. **(10 pts.) CODE EVALUATION, I.**

For each of the following, give the output of the code.

```
double[] d1 = { 1.0, 2.0, 3.0, 4.0, 5.0 };
double[] d2 = new double[d1.length * 2];
for ( int i = 0; i < d1.length; i++ )
{
    d2[i] = d1[i];
    d2[d2.length - i - 1] = d2[i];
}

for ( int i = 0; i < d2.length; i++ )
    System.out.print( d2[i] + " " );
```

Answer: this code inserts the values from `d1` into `d2`, front-to-back and then back-to-front, and prints them out as follows:

```
1.0 2.0 3.0 4.0 5.0 5.0 4.0 3.0 2.0 1.0
```

```
String[] words = { "Every", "good", "bird", "deserves", "feeding" };
for ( int w = 0; w < words.length; w++ )
{
    System.out.print( words[w] + " >=" );
    for ( int j = 0; j < words.length; j++ )
        if ( words[j].length() <= words[w].length() )
            System.out.print( " " + words[j]);
    System.out.println();
}
```

Answer: this code prints out each element of `words`, followed by all of those elements that are of length *less than or equal to* that of the first element:

```
Every >= Every good bird
good >= good bird
bird >= good bird
deserves >= Every good bird deserves feeding
feeding >= Every good bird feeding
```

4. **(5 pts.) CODE EVALUATION, II.**

Suppose we have the following two class definitions:

```
public class ClassA {
    protected int num1, num2;

    public ClassA( int i1, int i2 ) {
        num1 = i1;
        num2 = i2;
    }

    public boolean larger() {
        return ( num2 > num1 );
    }
}

public class ClassB extends ClassA {
    private int num3;

    public ClassB( int i1, int i2, int i3 ) {
        super( i1, i2 );
        num3 = i3;
    }

    public boolean larger() {
        boolean large1 = super.larger();
        boolean large2 = num3 > num2 ;
        return ( large1 && large2 ) ;
    }
}
```

For each of the `println()` statements below, give the output:

```
        ClassA a1 = new ClassA( 0, 0 );
        ClassA a2 = new ClassA( 0, 5 );
        ClassB b1 = new ClassB( 0, 0, 5 );
        ClassB b2 = new ClassB( 0, 6, 9 );
        ClassB b3 = new ClassB( 0, 6, 6 );

        System.out.println( a1.larger() );     \\ ANSWER:  false
        System.out.println( a2.larger() );     \\ ANSWER:  true
        System.out.println( b1.larger() );     \\ ANSWER:  false
        System.out.println( b2.larger() );     \\ ANSWER:  true
        System.out.println( b3.larger() );     \\ ANSWER:  false
```

5. **(10 pts.) CODE COMPLETION, I.**

Fill in the `main()` method in the class below so that when it runs it prints output (using `System.out.println()`) that looks like this:

```
0 1 2 3 4 5
1 2 3 4 5 0
2 3 4 5 0 1
3 4 5 0 1 2
4 5 0 1 2 3
5 0 1 2 3 4
```

For full points, your code must use **nested loops**, each of which is actually used to generate the output. (You may use whatever types of loops you choose.)

Answer: This pattern, which "rotates" the values, is most easily solved using the remainder function:

```
public class Main
{
    public static void main( String[] args )
    {
        for ( int i = 0; i < 6; i++ )
        {
            int num = i;
            for ( int j = 0; j < 6; j++ )
            {
                System.out.print( num + " " );
                num = ( num + 1 ) % 6;
            }
            System.out.println();
        }
    }
}
```

6. **(10 pts.) CODE COMPLETION, II.**

Fill in the class below. Add the `sumArray()` method that has been called from the class constructor. When run, this method should act as follows:

- If the arrays are of ***identical*** length, then it should return a new array of the ***same*** length, where each element is the sum of the elements at the same index in the input arrays. So, in the first call below, it would sum the first elements of the inputs, then their second elements, and so on, and output array `out1` would look like:

  `{ 2, 3, 5, 6, 8 }`

- If the arrays are of ***different*** lengths, then it will return an ***empty*** array, containing no data. (This is what would be returned for the second call, so `out2` would be empty.)

```
public class Main
{
     public Main()
     {
        int[] arr1 = { 1, 2, 3, 4, 5 };
        int[] arr2 = { 1, 1, 2, 2, 3 };
        int[] arr3 = { 1, 2 };

        int[] out1 = sumArrays( arr1, arr2 );
        int[] out2 = sumArrays( arr2, arr3 );
     }

     private int[] sumArrays( int[] a1, int[] a2 )
     {
         int[] output = new int[0];

         if ( a1.length == a2.length )
         {
             output = new int[a1.length];
             for ( int i = 0; i < output.length; i++ )
                 output[i] = a1[i] + a2[i];
         }
         return output;
     }
}
```

7. **(10 pts.) CODE COMPLETION, III.**

Fill in the class below. It should have two methods.

- The first, called `concatenate()`, should take in two `String` objects as input, and return a single `String` composed of the first input, followed immediately by the second input (with no spaces). Thus, calling `concatenate( "A", "B" )` will return `"AB"`.
- The second, called `remove()`, should take in a `String` and a single character, and return the result of removing every occurrence of that exact character from the `String`. Thus, calling `remove( "Test", 't' )` will return `"Tes"` (only removing the lower-case `'t'`), and calling `remove( "Test", 'x' )` will return `"Test"`, since it does not contain `'x'`.

```
public class Driver
{
    public String concatenate( String str1, String str2 )
    {
        String concat = str1 + str2;
        return concat;
    }

    public String remove( String str, char c )
    {
        String rem = "";
        for ( int i = 0; i < str.length(); i++ )
        {
            char current = str.charAt( i );
            if ( current != c )
            {
                rem = rem + current;
            }
        }
        return rem;
    }
}
```

8. **(15 pts.) CODE COMPLETION, IV.**

On the next page, there is code for a `Driver` class. It uses a class called `ColoredOval`, which you must write yourself. The class you write should work as follows:

(a) It should extend the basic `Oval` class.

(b) It should have a constructor that takes in its (x, y) location, its *width* and *height* in pixels, and a `Color`. It then creates a graphical oval object with the given location and size, colored the input `Color`. Thus, when the `Driver()` constructor finishes running, the window will contain a red circular object.

(c) It should have a method called `recolor()` that changes the color of the object at random to either *blue* or *green*. Thus, whenever the button marked "Change" is pressed, the `Driver` code will cause the on-screen object to change color randomly.

(d) It should have a method called `backToOriginal()` that returns it to its original color. Thus, whenever the button marked "Return" is pressed, the `Driver` will cause the object to become red again.

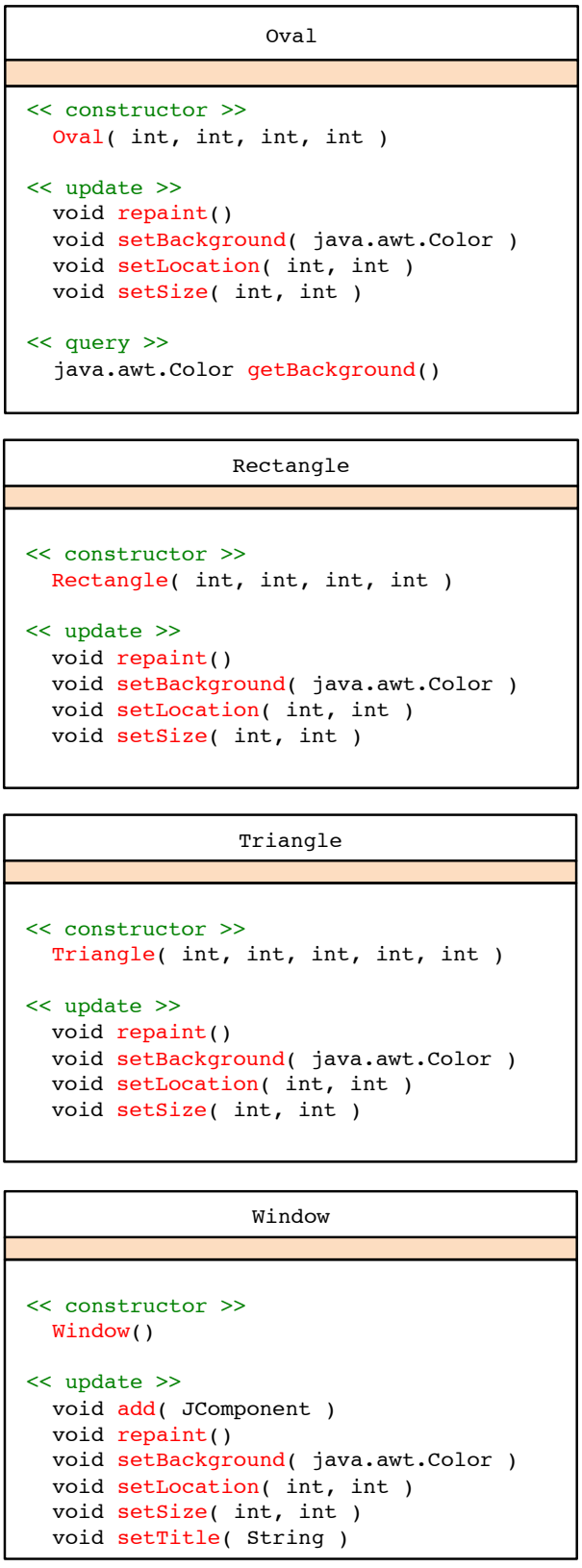
(e) It should have a method called `getOriginalColor()` that returns its original color. Thus, whenever the last `println()` statement in the `Driver` code executes, we will see information indicating that the original color is red.

(f) It should **over-ride** the existing `setBackground()` method in the parent class so that it does nothing. This way, only the previous methods can be used to change the color.

Note: class diagrams for graphical classes appear on the last page of the exam.

```
import java.awt.Color;

public class Driver
{
    private ColoredOval colored;

    public static void main( String[] args )
    {
        Driver d = new Driver();
        d.makeColoredOval();
    }

    // creates an Oval, and then changes its color 10 times
    private void makeColoredOval()
    {
        Window window = new Window();
        window.setSize( 500, 500 );
        window.setBackground( Color.white );
        ActionButton button = new ActionButton( this );
        button.setBounds( 200, 450, 100, 30 );
        button.setText( "Change" );
        window.add( button );

        colored = new ColoredOval( 200, 200, 100, 100, Color.red );
        window.add( colored );
    }

    // responds to button press by changing ColoredOval
    public void handleAction( ActionButton button )
    {
        if ( button.getText().equals( "Change" ) )
        {
            button.setText( "Return" );
            colored.recolor();
        }
        else
        {
            button.setText( "Change" );
            colored.backToOriginal();
        }

        Color original = colored.getOriginalColor();
        System.out.println( "Oval color is now: " + colored.getBackground() );
        System.out.println( "Original color was: " + original );
    }

}
```

```
import java.awt.Color;

public class ColoredOval extends Oval
{
    private Color originalColor;

    public ColoredOval( int x, int y, int w, int h, Color c )
    {
        super( x, y, w, h );
        originalColor = c;
        super.setBackground( originalColor );
    }

    public void recolor()
    {
        int num = (int)( Math.random() * 2 );
        if ( num == 0 )
        {
            super.setBackground( Color.blue );
        }
        else
        {
            super.setBackground( Color.green );
        }
    }

    public void backToOriginal()
    {
        super.setBackground( originalColor );
    }

    public Color getOriginalColor()
    {
        return originalColor;
    }

    public void setBackground( Color c )
    {
        // does nothing
    }
}
```


Oval
<< constructor >>
Oval(int, int, int, int)
<< update >>
void repaint()
void setBackground(java.awt.Color)
void setLocation(int, int)
void setSize(int, int)
<< query >>
java.awt.Color getBackground()
Rectangle
<< constructor >>
Rectangle(int, int, int, int)
<< update >>
void repaint()
void setBackground(java.awt.Color)
void setLocation(int, int)
void setSize(int, int)
Triangle
<< constructor >>
Triangle(int, int, int, int, int)
<< update >>
void repaint()
void setBackground(java.awt.Color)
void setLocation(int, int)
void setSize(int, int)
Window
<< constructor >>
Window()
<< update >>
void add(JComponent)
void repaint()
void setBackground(java.awt.Color)
void setLocation(int, int)
void setSize(int, int)
void setTitle(String)