# FCPS
# Java
# Packets

## Unit Three – GUIs & I/O

**January 2019**

**Developed by Shane Torbert**
**edited by Marion Billington**
**under the direction of Gerry Berry**
**Thomas Jefferson High School for Science and Technology**
**Fairfax County Public Schools**
**Fairfax, Virginia**

**Contributing Authors**

The author is grateful for additional contributions from Michael Auerbach, Marion Billington, Charles Brewer, Margie Cross, Cathy Eagen, Anne Little, John Mitchell, John Myers, Steve Rose, Ankur Shah, John Totten, and Greg W. Price.

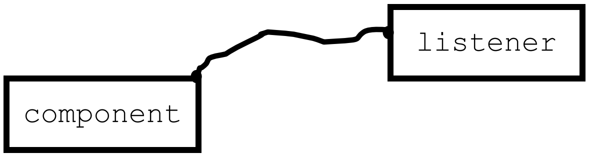
The students' supporting web site can be found at http://academics.tjhsst.edu/compsci/CSweb/index.html
The teacher's (free) FCPS Computer Science CD is available from Stephen Rose (srose@fcps.edu)

**License Information**

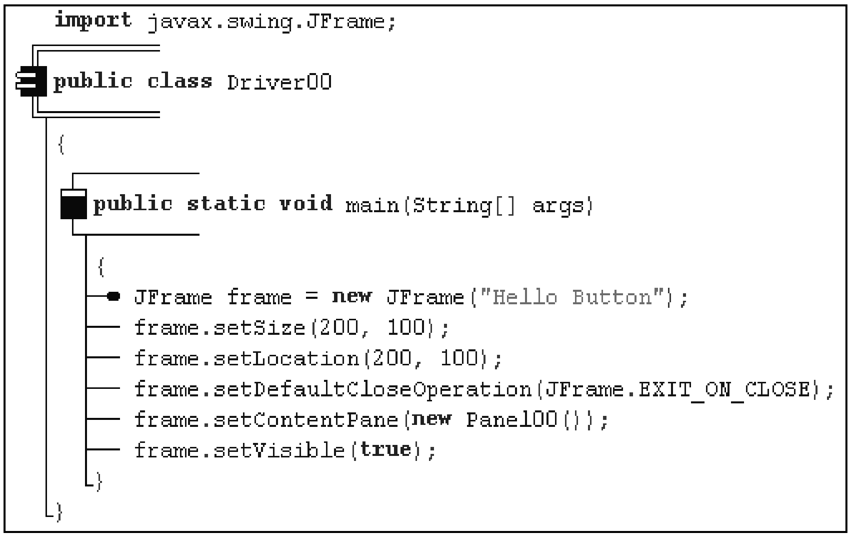
# Java Instruction Plan—Unit Three

# Discussion
## Panels, GUI Components, and Listeners

A Graphical User Interface (a GUI, pronounced “gooey”) allows users to point and click on on-screen "buttons." Almost all computers today are operated through GUIs, but once upon a time people used keyboards, punchcards, or even dials and toggle switches. A GUI is quite hard to program, unless you are in an *object-oriented* language, in which case it is fairly easy. That's because an object oriented language takes advantage of *abstraction*, *inheritance*, and *encapsulation*. For example, each object in a GUI, such as the buttons, textfields, labels, panels, and frame, "knows" its own attributes and methods—that is encapsulation. Like the Polkadot and Ball classes in Unit 2, each GUI object even knows how to draw itself. Each object inherits some of its methods from higher up the hierarchy, overrides some methods, and adds new methods. The hierarchy proceeds from abstract to concrete, with the programmers assigning powers at the appropriate level. For example, abstractly considered, all JComponent objects can be assigned ActionListeners. You, the programmer, work with the concrete objects when you assign a specific button to a specific listener. (You did the same thing in Unit 2, when you assigned a timer to a specific listener.) You also, as the last programmer in the chain, get to say exactly what happens when each button is clicked.



The structure of the applications in this unit is exactly the same as in Unit 2. The driver creates a frame, puts the panel on the frame, and displays the frame. The panel has the buttons, labels, text boxes, and even other panels. Visually, a GUI is not a whole lot different from straight graphics. However, in a GUI the program reacts to something the user does. Unlike programs in Units 1 and 2, our GUI will wait patiently until a button is clicked, then it springs forth and does its thing. Best of all, we don't have to know the details of how all that works. Someone else programmed buttons and listeners for us.



```
import javax.swing.JFrame;
public class Driver00
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Hello Button");
      frame.setSize(200, 100);
      frame.setLocation(200, 100);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setContentPane(new Panel00());
      frame.setVisible(true);
   }
}
```

Just for fun, here is the hierarchy, including interfaces, for a JButton:

java.lang.Object
└ java.awt.Component
  └ java.awt.Container
    └ javax.swing.JComponent
      └ javax.swing.AbstractButton
        └ javax.swing.JButton

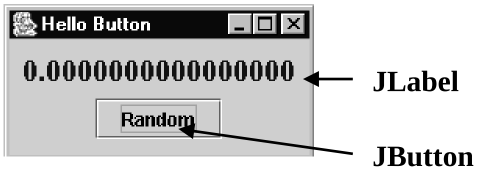
All Implemented Interfaces:

Accessible, ImageObserver, ItemSelectable, MenuContainer, Serializable, SwingConstants

# Lab00
## Hello Button

***Objective***
Understand frames, panels, and GUI components.

***Background***
The drivers in Unit 3 are exactly the same as those in Unit 2. The panel will have GUI components like this:



***Specification***
Create Unit3\Lab00\Driver00.java and Unit3\Lab00\Panel00.java. Enter the source code for the driver. Enter the source code for the resource class (the panel) as shown below, then compile and run the program.

```
 4 import javax.swing.*;
 5 import java.awt.*;
 6 import java.awt.event.*;
 7 public class Panel00 extends JPanel
 8   {
 9   private JLabel label;      //reference in a field
10   public Panel00()
11    {
12     setLayout(new FlowLayout());
13
14     label = new JLabel("0.0000000000000000");
15     label.setFont(new Font("Serif", Font.BOLD, 20));
16     label.setForeground(Color.blue);
17     add(label);
18
19     JButton button = new JButton("Random"); //local
20     button.addActionListener(new Listener());
21     add(button);
22    }
23   private class Listener implements ActionListener
24    {
25    public void actionPerformed(ActionEvent e)
26      {
27       double x = Math.random();   //local variable
28       label.setText("" + x);
29      }
30    }
```

Lines 4-6: import the swing, awt, and event classes.

Line 9: The reference to `label`, which is empty, is *declared* in a private *field*. We need to declare the reference here because we want the listener to access `label` (in line 28).

Lines 10-22: The constructor.

Lines 14-17: We *instantiate* the label, set its font and color, and add it to the panel.

Lines 19-21: Instantiate a button, register its listener, and add it to the panel.

Line 23: Listener implements the ActionListener interface, meaning that actionPerformed must have code.

Line 28: `("" + x)` converts a number into a string, because `setText` requires a string argument.

The *scope* of a variable is the region of the code in which that variable is accessible. This kind of accessibility has nothing to do with `public` and `private`, but everything to do with curly braces. We need to *declare* the `JLabel label` in a field on Line 9 and *instantiate* it on Line 14 because we want to access it on Line 28. In contrast, we declared and instantiated `JButton button` on Line 19, restricting its scope to lines 19-21, because that is the only place we need to access it. Similarly, the `double x` in Line 27 has a scope that is restricted to Lines 27-28, because that is the only place where we want to use it. We may say that the `label` in Line 9 is *global* in scope and that the `button` and the `x` are *local* in scope. The blue lines down the side in jGrasp show the scope of the different variables.

***Sample Run*** `Unit3Lab00.jar`

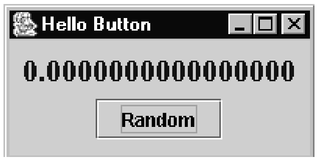
# Exercises

## Lab00

1) What three steps do you have to do to create buttons?

2) `add(label)` adds a label to the _______________.

3) There are at least four GUI components showing below. Label them all.



4) A fifth GUI component is hidden, but you know it is there. What is it?

5) What do you have to do to program the action of a button?

6) What interface does every button's listener implement? ____________________________

7) What method is in that interface? _________________________________

8) "To implement an interface" means to write concrete methods for all the ___________________ methods in the interface.

9) Give the range of values that the following will generate:

   a. `Math.random();` ____________________

   b. `Math.random() * 10;` _____________________

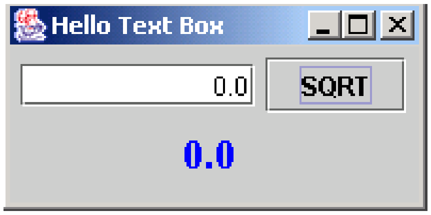
   c. `Math.random()  * 100;` _______________________

   d. `Math.random() * 6;` ______________________

# Discussion
## GUI components, strings, numbers

The initial text that appears in any label, button, or text box can be specified through the object's constructor.

```
private JLabel label;
private JTextField box;
public Panel01()
{
  label = new JLabel("0.0");
  JButton button = new JButton("SQRT");
  box = new JTextField("0.0", 10);
```



In the last command, the textField named "box" will display 0.0, be right justified, and be 10 characters wide.

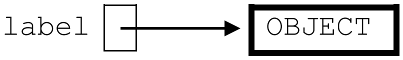
Each GUI component has formatting commands which must be called individually. There is no way to format all your components using one command. The API lists the formatting commands for each kind of component.

Notice the arrows above. In these labs, we often take two steps to create a reference to an object. First, in the private fields of the class, we *declare* a *reference* without an object:

```
private JLabel label;
```

Later, in the constructor, we will point that reference to the object that we instantiate. As always, we *instantiate* an *object* by calling the class's *constructor*, including the keyword `new`, like this:

```
label = new JLabel("");
```



We declare and instantiate in separate steps whenever we need to access the private field's reference in several methods, i.e. when the scope of the field must be global. We saw this in Lab00 at lines 9, 14, and 28.

---

To a Java compiler, "6" is different from 6, which is different from 6.0. "6" is a string, 6 is an integer, and 6.0 is a double, and each representation of 6 is stored in a completely different way. We humans want to convert each kind of 6 into the other kinds. In Lab01, we will want to retrieve a number from the textField. We use the method `getText`, which returns a string. Then we turn that string (the "6") into a double (the 6.0) using the `Double.parseDouble` method.

```
double d = Double.parseDouble( box.getText() );
```

So now we have stored the string from the textField as a `double` value in `d`. Usually in these labs, we then do some sort of arithmetic with `d`, often saving the result in a new `double` variable. The third step is to display the result on the screen. We use `setText` to display the result on the label, but first we must change the number back into a string. We change the number into a string with `(""+d)`, which is a Java shortcut that returns the *string representation* of the number. The whole command is

```
label.setText("" + d);
```

# Lab01
## Hello Text Box

***Objective***
Fields vs. local variables

***Background***
Lines 9-10: These two declarations create private fields because the fields need to be accessed later in the listener.

Line 13: FlowLayout is the simplest GUI layout. The first component added to the panel is placed at the center top. Additional components are placed to the right, or on the next line. Be sure to drag the corner of the frame and watch how the flow works.

Line 15-26. Notice the different components are instantiated, formatted, and added to the panel. Components appear in the order in which they are added.

Line 19: This button is declared locally because it does not need to be accessed in the listener, unlike the JLabel and the JTextField which are declared in the fields.

```
 4  import javax.swing.*;
 5  import java.awt.*;
 6  import java.awt.event.*;
 7  public class Panel01 extends JPanel
 8   {
 9    private JLabel label;        //field
10    private JTextField box;       //field
11    public Panel01()
12      {
13      setLayout(new FlowLayout());
14
15      box = new JTextField("0.0", 10);
16      box.setHorizontalAlignment(SwingConstants.RIGHT);
17      add(box);
18
19      JButton button = new JButton("SQRT");    //local
20      button.addActionListener(new Listener());
21      add(button);
22
23      label = new JLabel("0.0");
24      label.setFont(new Font("Serif", Font.BOLD, 20));
25      label.setForeground(Color.blue);
26      add(label);
27      }
28    private class Listener implements ActionListener
29      {
30       public void actionPerformed(ActionEvent e)
31       {
32           //get the number
33           //take the square root
34           //display it
35       }
36      }
37   }
```

***Specification***
Create Unit3\Lab01\Driver01.java. Create an appropriate driver.

Open Unit3\Lab01\Panel01.java. Type in the resource class above. Implement the method `actionPerformed` to find square roots. What happens if you type a negative number into the text box?

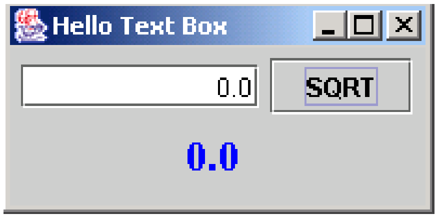
***Extension***
Edit your program to handle negative numbers correctly, i.e., $\sqrt{-25}$ returns 5i. You will need an if-statement.

***Warning***
A common error is to write Line 23 as `JLabel label = new JLabel("0.0");` Write Line 23 with this mistake. Compile and run. Now you have declared two JLabels, but instantiated only one of them. Explain what `null pointer exception` means.

***Sample Run*** `Unit3Lab01.jar`

# Exercises
## Lab01

1) There are at least five GUI components showing below. Label them all.



2) Describe a “string” in Computer Science.

3) Describe a “double” in Computer Science.

4) How do you get a string that is in a textField?

5) How do you change that string into a double?

6) How do you change a double into a string, as when you `setText` a numerical answer into a label?

7) Define and/or explain every underlined term.:

```
      double d  =  Double.parseDouble( textField.getText() );
```

*Complete this scavenger hunt using the current on-line Java API. (In jGrasp, go to Help|Java API.)*

8) What is the lowest common ancestor of JButton, JLabel, and JTextField? ___________________________

9) How many methods does the ActionListener interface specify? ___________________________

10) What package is ActionEvent in? ___________________________

11) How many classes extend JButton? ____________________

12) How many constructors does JButton define? ___________________________

13) How many methods does JButton define? ___________________________

14) JButton inherits a `setText` method. When would you ever want to use a button's `setText` in the middle of a program?

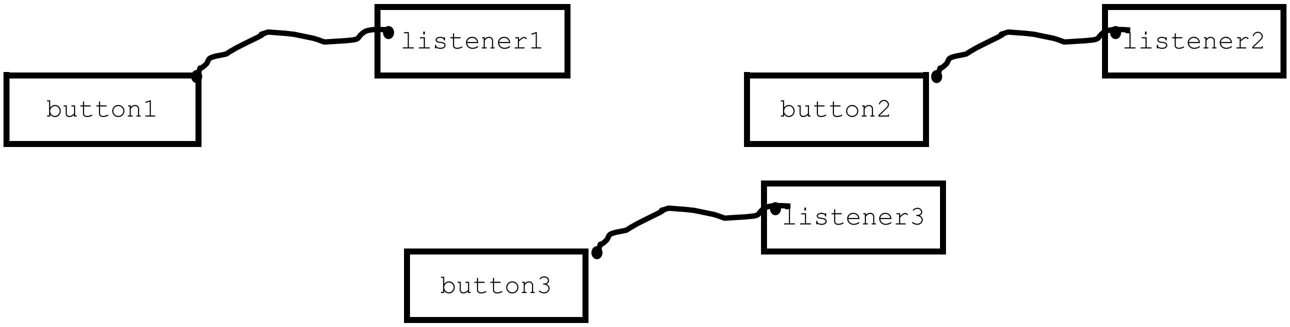
15) How many constructors does JTextField define? ___________________________

16) Write the code to instantiate a JTextField with "Enter name" in the field, and the field is 20 spaces wide.

# Discussion

## Multiple Buttons

The GUI model we have seen thus far can be applied to any number of buttons and listeners.



Simply declare a separate listener class for each different action that has to be performed. Register listener objects with buttons appropriately.

```
button1.addActionListener(new Listener1());
button2.addActionListener(new Listener2());
button3.addActionListener(new Listener3());
```

The code above will work, but the names are not informative. You should use informative names, such as:

```
randomNumber.addActionListener(new RandomListener());
reciprocalButton.addActionListener(new ReciprocalListener());
quitButton.addActionListener(new QuitListener());
```

As the code below says, each private listener class *implements* the ActionListener interface. That interface specifies exactly one abstract method, `actionPerformed`. That means that each Listener code must define its own `actionPerformed` method to specify what happens on each button-click.

Be careful! The placement of curly braces is critical to ensure the proper *scope* of the listeners. Each of the listener classes must be placed at the level of the panel.

Sadly, a common error is to forget the curly brace that closes the listener class, with the unfortunate result that the second listener is defined entirely inside the definition of the first listener. Not good.

```
public class Panel02 extends JPanel
   {
    //fields--for objects that must be accessed later
    public Panel02()           //constructor
       {
         //format the panel,
         //instantiate and format the GUI components,
         //register the listeners,
         //add the GUI components.
       }
    private class Listener1 implements ActionListener
       {
       public void actionPerformed(ActionEvent e)
         {
          //do something
         }
       }
    //another listener
    //a third listener
    //a fourth listener
   }
```

# Lab02
## Multiple Buttons

***Objective***
GUIs and listeners.

***Background***
Remember that GUI labels and textFields always display Strings, even though they may look like numbers to you. When you `getText` from a label, you will be getting a String. If you want to do arithmetic on that String, you will need to parse it, to convert it either to an int or a double.

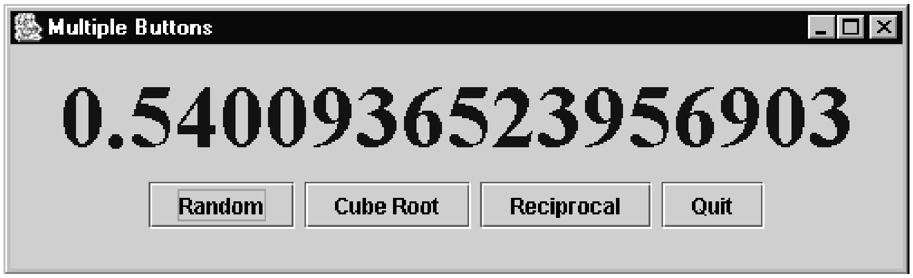
Going the other way, when you `setText` a number to a label, you must first convert the number into its *string representation*. In Java, we do that with `""+`

Some static (class) methods in the Math class that you should know:

| **Math function** | **Example command** |
|---|---|
| Absolute value | `x = Math.abs(y);` |
| Square root | `x = Math.sqrt(y);` |
| Square | `x = y * y;` or `x = Math.pow(y, 2);` |
| Cube root | `x = Math.pow(y, 1.0/3.0);` |
| Set x equal to 1 | `x = Math.pow(y, 1/3);` **//due to integer division, 1/3 → 0** |
| Raise to a power | `x = Math.pow(2, 10);` **//Recall that $2^{10}$ = 1024** |
| Sine, y in degrees | `x = Math.sin(y * Math.PI / 180.0);` |
| Cosine, y in degrees | `x = Math.cos(Math.toRadians(y));` |

***Specification***
Create Unit3\Lab02\Driver02.java. Create an appropriate driver. Make sure to tell the frame to add a panel object of type Panel02. Set the size of the frame so that it looks like that of the .jar demo.

Create Unit3\Lab02\Panel02.java. Reverse engineer the panel based on the .jar demo. Notice that there is no textField; there is only one label. Your code must get the value from the label, do an operation, then put the value back on the label. (We are using the label as if it were a variable.) Since you have four buttons, you will have to have four listeners. To make the Quit button work, check the System class in the current Java API for some hints.



***Sample Run***
`Unit3Lab02.jar`

# Exercises
## Lab02

1) `label.setText(number)` gives an error message. What is the error, and how do you fix it?

2) How do you change a string into a double?

3) Define and/or explain every underlined item.

```
private class Handler implements ActionListener
 {
     public void actionPerformed(ActionEvent e)
     {
          int num  =  Integer.parseInt( text.getText() );
          int result  =  num * num;
          label2.setText ("The square of the number is " + result);
     }
 }
```

Vocabulary practice:

3a) In the code above, `ActionListener` is a(n) ___________________

3b) In the code above, `actionPerformed` is a(n) ___________________

3c) In the code above, `Integer` is a(n) ___________________

3d) In the code above, `parseInt` is a(n) ___________________

3e) In the code above, `text` is a(n) ___________________

3f) In the code above, `getText` is a(n) ___________________

3g) In the code above, `int` is a(n) ___________________

3h) In the code above, `result` is a(n) ___________________

3i) In the code above, `label2` is a(n) ___________________

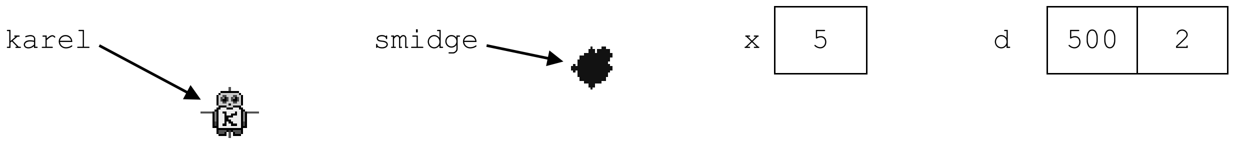
3j) In the code above, `setText` is a(n) ___________________

# Discussion
## Primitive Data Types

A *data type* defines a specific kind of data, e.g., numbers, Strings, or Robots, and how a computer internally stores and manipulates that data. A *class*, either imported or user-defined, defines some specific data type. Java also provides several *primitive data types* including `int` (for integers) and `double` (for decimal numbers). Primitive data types do not contain any methods. Useful integer methods are instead defined in a *wrapper class*. The integer wrapper class is called Integer and the decimal wrapper class is called Double. Thus, we have `Integer.parseInt` and `Double.parseDouble`. Other useful methods are in the API.

Objects are accessed by references, but primitives store their data directly. The convention in Java is that names of classes start with a capital letter and names of primitives are lower case, reserved words.

| | | | |
|---|---|---|---|
| Robot karel = **new** Robot(); | Turtle smidge = **new** Turtle(); | **int** x = 5; | **double** d = 5.00; |



| | | | |
|---|---|---|---|
| `Robot` is a class,<br>`karel` is a reference | `Turtle` is a class,<br>`smidge` is a reference | primitive type: **`int`**<br>`x` stores 5 | primitive type: **`double`**<br>`d` stores 5.00 |

One tricky part about `int`s and `double`s is *casting*, which means changing the data type from one to the other. Sometimes Java casts automatically and sometimes you must cast explicitly. Suppose your code is:

```
int x = 4;          //creates room for an integer, calls it "x", and stores 4 in it.
double y = 0;       //creates room for a double, calls it "y", and stores 0.0 in it.
...
y = (double)x;      //it works: the code casts 4 explicitly, which becomes 4.0 and is stored in y.
y = x;              //it works:  the integer 4 is promoted to 4.0 automatically.
```

However,

```
x = y;              //illegal, due to loss of precision.  The 0.0 does not become 0 automatically.
x = (int)y;         //it works:  you must explicitly tell the computer to truncate the decimal.
```

Presumably, the Java creators decided that 4 can become 4.0 without risk, but that decimals shouldn't lose their decimal parts without the programmer explicitly saying that it is okay.

If you copy and paste code too much you may accidentally apply `parseInt` or `parseDouble` when you need the other one. Always look at the type of the variable and choose appropriately.

```
int x = Integer.parseInt("5");
double y = Double.parseDouble("5.0");
```

Don’t forget how to change numbers into strings: `label.setText("" + number);`

Java is a strongly typed language, meaning that Java is very particular about requiring the correct data type, either `int`, `double`, `String`, or some other object. A large part of the compiler’s job is to guarantee *type-safe* code. Some other languages, such as Python, do not care so much about producing type-safe code.

# Discussion

## Integer Division and Modulus

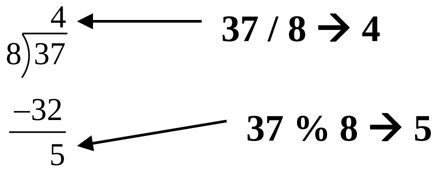
In Java, when you **divide** two integers you will always get an integer quotient. The remainder, if any, is ignored and thrown away. For example:

```
int x = (10 * 6 + 7 - 30) / 8;                x | 4 |
```

Sometimes, finding the remainder is quite useful. The remainder is calculated by the **modulus** operator, whose symbol in Java is **%**. For instance:

```
int y = (10 * 6 + 7 - 30) % 8;                y | 5 |
```

To understand **integer division** and **modulus**, think back to long division. For example, 37 ÷ 8



Notice that the / does not round up. Any remainder is *truncated*. Truncated means discarded. If you did 9999/10000, the result would equal 0. If you did 9999%10000, the result would equal 9999.

The Hailstone algorithm requires you to determine if an integer is even. A good way to do this is to check if the remainder of the integer divided by two equals zero. Write that line of code here:

# Lab03

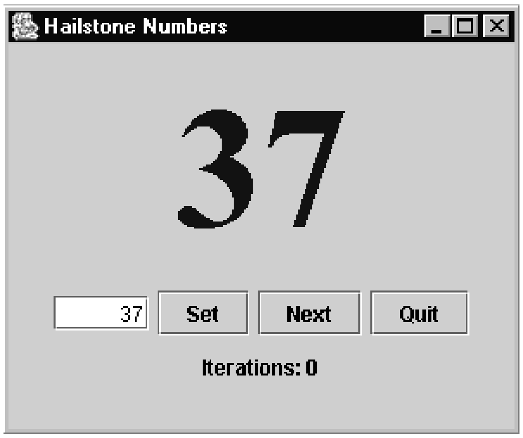
## Hailstone Numbers

***Background***
Here is the Hailstone algorithm: start with a random integer from 1 to 100. If the number is even, divide it by two to generate the next number. If the number is odd, multiply it by three and add one to generate the next number. Keep generating in that way. What happens? The answer is that you get a sequence of hailstone numbers, which you can see in action by running `Unit3Lab03.jar`.

***Specification***
Create Unit3\Lab03\Driver03.java. Create a driver.

Open Unit3\Lab03\Panel03.java. Complete the definition of each listener's `actionPerformed`. What does each button do? Unlike Lab02, we have two *variables* as *fields* of this panel, `number` and `count`. You are to do the math using the variables, then display their values on the labels.

***Sample Run***
`Unit3Lab03.jar`
Be sure to play with the demo.



37

Iterations: 0

*Extension for math geniuses*: Every sequence of Hailstone numbers seems to end in 4, 2, 1, but no one has proved it to be so. If you come up with a proof, your name will be honored among mathematicians.

# Exercises

## Lab03

1) Practice integer division, modulus division, and decimal division. Remember the order of operations.

| | | | |
|---|---|---|---|
| 14 / 5 → | 14 % 5 → | 14.0 / 5.0 → | 3 + 19 % 6 – 4 → |
| 15 / 5 → | 15 % 5 → | 15.0 / 5 → | 1 + 8 % 2 → |
| 16 / 5 → | 16 % 5 → | 16 / 5.0 → | 2 / 3 * 3 → |
| 17 / 5 → | 17 % 5 → | 17 / 5. → | 1234 % 10 / 1 → |
| 3 / 5 → | 3 % 5 → | 3 / 5. → | 1234 % 100 / 10 → |
| 0 / 5 → | 0 % 5 → | 0.0 / 5 → | 1234 % 1000 / 100 → |

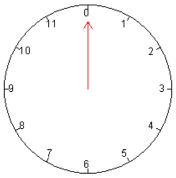
2) **int** amount = Integer.parseInt (“143”);
**int** quarters = amount / 25;
amount = amount % 25;
**int** dimes = amount / 10;
amount = amount % 10;
**int** nickels = amount / 5;
**int** pennies = amount % 5;
**int** numCoins = quarters + dimes + nickels + pennies;

| | |
|---|---|
| amount | □ |
| quarters | □ |
| dimes | □ |
| nickels | □ |
| pennies | □ |
| numCoins | □ |

3) Given total number of seconds, write assignment statements to store the number of minutes and the number of extra seconds. For example, if `totalSec = 500`, then `min = ` _____ and `sec = ` _____

4) Given the total number of seconds, write assignment statements to store the number of hours, minutes, and seconds. For example, if `totalSec = 19230`, then `hr = 5`, `min = 20`, and `sec = ` _____

5) Think of a clock face. What is 8:00 plus 5 hours? ____



In mathematical terms, clock arithmetic is addition modulus 12.
That is, 8 + 5 = 1 because (8+5) % 12 = 1.
What is 8 + 20? ______ What is 3 + 15? ______
What is 8 + 24? ______ What is 5 + 20? ______
What is 8 + 39? ______ Explain how you would multiply, e.g. 8 * 3 = 0

6) Evaluate. Remember the order of operations.

a. `23 / 5 – 10 % (5 – 1) * 3 + 2 / 6` ______
b. `3 % 4 + 5 * 2 / 33 - 3 + 8 * 3` ______
c. `6 + 6 % 6 + 6 / 6 + 6 * 6 – 6` ______
d. `(8 % 4 == 0) && (12 % 4 == 0)` ______

# Discussion
## The Black Box

Most of us have no idea how buttons and listeners actually work. They are like a *black box* to us, in that their private data and their implementation code are *hidden* and *encapsulated*. The client, often another programmer, uses black box objects without knowing how they are doing what they are doing. Furthermore, we don't care how they work! (If you really want to know how buttons and listeners work, ask your teacher how to view the Java *source code*.)

In Lab04, you get a chance to do your own black box programming. Driver04 looks exactly like all the other drivers. It hasa Panel04, which has already been written and compiled for you, but is shown below. Do not type it in!

Lines 4-6: Import some mysterious classes.

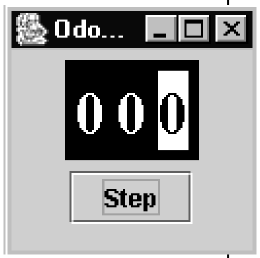
Line 9: declares an empty reference in the field.

Lines 14-15: The code points the reference to the odometer object and adds that object to Panel04. What is an odometer object? The panel doesn’t know!

Lines 17-19: instantiate a local JButton, as usual.

Line 25: When the button is clicked, the `update` method in the Odomoter object is called. What is `update`? At this point, we don't know. All we know is that it is an instance method in the Odometer class.

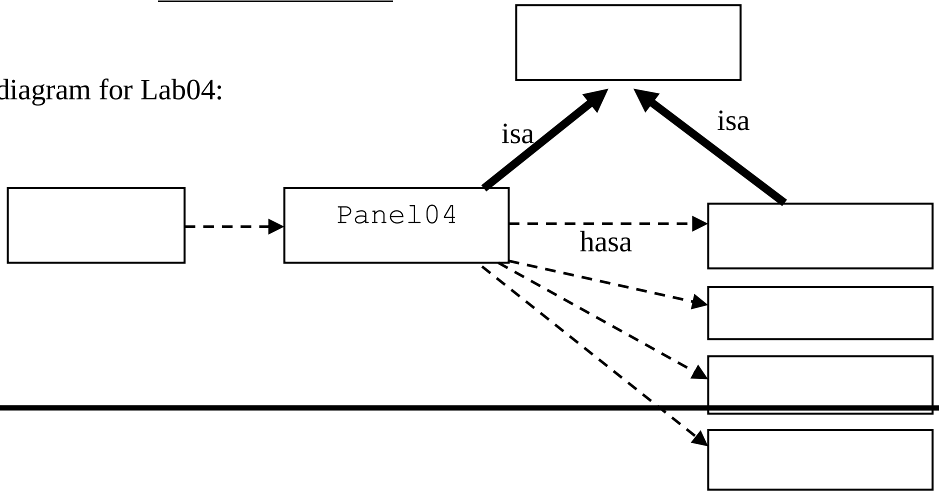
```
 4  import javax.swing.*;
 5  import java.awt.*;
 6  import java.awt.event.*;
 7  public class Panel04 extends JPanel
 8   {
 9    private Odometer odometer;
10    public Panel04()
11     {
12      setLayout(new FlowLayout());
13
14      odometer = new Odometer();
15      add(odometer);
16
17      JButton button = new JButton("Step");
18      button.addActionListener(new Listener());
19      add(button);
20     }
21    private class Listener implements ActionListener
22     {
23      public void actionPerformed(ActionEvent e)
24       {
25         odometer.update();
26       }
27     }
28   }
```



Your job, should you decide to accept it, will be to program the Odometer class.

Draw the CSD scope lines in the code above. Give an example of a *global* reference: ____________ Give an example of a *local* reference: _________________

Fill in the UML diagram for Lab04:

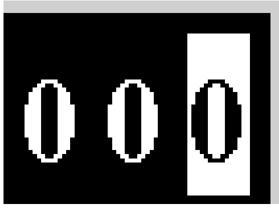


# Lab04
## Odometer

***Objective***
Look inside a black box. Modulus and Integer Division ( % and / )

***Background***
The shell for the Odometer class is shown below.

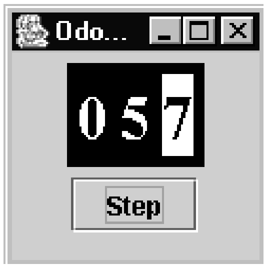
```
import javax.swing.*;
import java.awt.*;
public class Odometer extends JPanel
{
  /****************************/
  /* Declare references in    */
  /*      3 fields.           */
  /*                          */
  /* Declare an int variable  */
  /*      in 1 field          */
  /****************************/

  public Odometer()
   {
    /*******************************/
    /* This is the constructor.    */
    /* Set the layout, set the     */
    /*    background.              */
    /* Instantiate all objects and */
    /* set their properties.       */
    /*******************************/
   }
  public void update()
   {
    /***************************/
    /*                         */
    /* Increment the counter,  */
    /* set the labels.         */
    /*                         */
    /***************************/
   }
  }
```



An odometer is an object that displays numbers. Clearly, the Odometer class should extend the JPanel class. Since Odometer isa JPanel, set the layout. Set the background to black. Add the GUI components to make the odometer panel look like an odometer. Which kind of components do you need? How many?

Default labels are “clear”. You will have to make them “opaque.” You'll also have to set the background and the foreground colors. Look at the Java API.

An odometer also keeps count. You will have to add a counter to the odometer.



Someone using Panel04 will click on the "step" button. That action causes the odometer’s `update` method to be called. When `update` is called, it increments the counter and displays the result in the labels. In the case shown, the counter variable is storing 57. To display the result in the labels you will have to use / and %.

Once again we have an example of encapsulation at work. Updating is something Odometer does; button-clicking is something Panel04 does; and the frame holds it all.

***Specification***
Open Unit3\Lab04\Odometer.java. Complete the implementation of the Odometer class as described above.

Unit3\Lab04\Panel04.class. This file has been provided in compiled form. Don’t overwrite it!

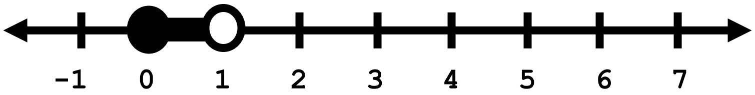
Open Unit3\Lab04\Driver04.java. Compile and run.

***Sample Run***
`Unit3Lab04.jar`

# Discussion
## Random Numbers

The next few labs take advantage of the computer's ability to generate *random numbers*. (Theoretically, a computer is able only to generate a pseudorandom number, not an actual random number. But pseudorandom numbers are good enough for most purposes.) Java has two ways to generate random numbers, using the method `Math.random` or using the Random class. We will learn the first way only.

`Math.random` returns a pseudorandom decimal number that is greater than or equal to zero and strictly less than one. On the number line we graph it as:



Expressed in mathematical notation, the random number is generated on the interval [0, 1), or 0 <= x < 1.0. Notice 1.0 is excluded from the possible values that can be returned from a call to `Math.random`. Put another way, the greatest possible value returned from `Math.random` is 0.999999. . . .

What if we want random numbers on other intervals? We can generate them by multiplying and adding the output of `Math.random` as needed. You may visualize multiplying as "stretching" the output set of random numbers and adding as "shifting" the output set.

| If you want: | Then do: |
|---|---|
| a `double` [0, 6) or 0 <= x < 6.0 | `Math.random() * 6;` |
| a `double` [1, 7) or 1 <= x < 7.0 | `Math.random() * 6 + 1;` |
| a `double` [25, 75) or 25 <= x < 75.0 | `Math.random() * 50 + 25;` |
| a `double` [10, 100) or 10 <= x < 100.0 | `Math.random() * 90 + 10;` |
| a `double` [-6, 6) or -6 <= x < 6.0 | `Math.random() * 12 - 6;`<br>This formula is what we used to generate dx and dy in the Ball class in Unit 2. |

By studying these examples, you may be able to arrive at the general formula for using `Math.random`:

```
   double d = Math.random() * RANGE_OF_VALUES + LOWEST;
```

Since `Math.random` returns a `double`, how can we generate integers? In that case you must *cast* the `double` into an `int`, using `(int)` as shown on the right side:

| If you want: | Then do: |
|---|---|
| an `int` in {0, 1, 2, 3, 4, 5} | `(int)(Math.random() * 6);` |
| an `int` in {1, 2, 3, 4, 5, 6} | `(int)(Math.random() * 6 + 1);` |
| an `int` in {25, 26, 27, . . . 74} | `(int)(Math.random() * 50 + 25);` |
| an `int` in {10,11,12,. . . 99} | `(int)(Math.random() * 90 + 10;` |
| an `int` in {-6,-5,-4, . . .5} | `(int)(Math.random() * 12 – 6);` |

The general formula for obtaining an integer from `Math.random` is:

```
   int x = (int)(Math.random() * (HIGHEST - LOWEST + 1) + LOWEST);
```

Be careful! Do not forget the extra parentheses around the expression, or `(int)` will truncate `Math.random`, which will generate 0. You don't want to go to all that trouble just to generate a zero.

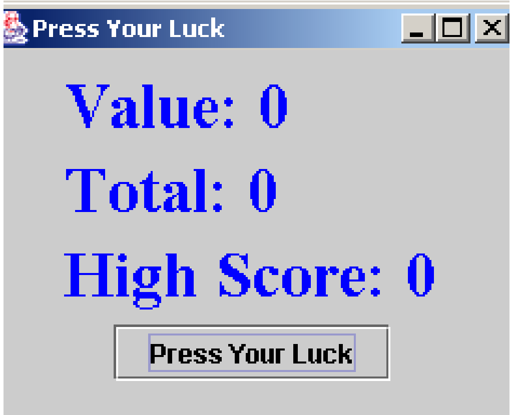
# Lab05
## Press Your Luck

***Objective***
Random numbers.

***Background***
Play with the demo at `Unit3Lab05.jar`. When do Value and Total change? When does High Score change?

How many labels do you need? How many buttons? When you click the button, what happens?

This lab is organized with a driver, a panel, and a display. The display encapsulates the three labels. The panel holds the display and the button. Locate these elements at the right.



Value: 0
Total: 0
High Score: 0

```
 7 public class Panel05 extends JPanel
 8 {
 9  private Display05 display;
10  public Panel05()
11   {
12    setLayout(new FlowLayout());
13
14    display = new Display05();
15    add(display);
16
17    JButton button = new JButton("Press Your Luck");
18    button.addActionListener(new Listener());
19    add(button);
20  }
21  private class Listener implements ActionListener
22  {
23   public void actionPerformed(ActionEvent e)
24    {
```

How is Panel05 designed? Notice that Line 9 declares a reference that is empty at this point. Line 14 instantiates the Display object and points the reference to it. Line 15 adds the display to the panel, exactly as the Odometer was instantiated and added in Lab04.

Lines 17-29: code the typical steps to create a working button.

Lines 21-28: you need a listener, of course.

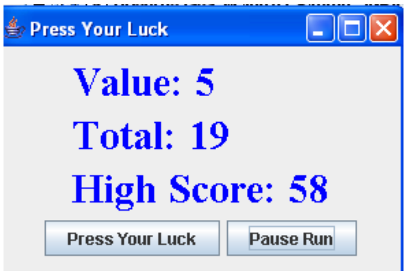
***Specification***
Open Unit3\Lab05\Display05.java. Use the fields `value` and `total`. Create a field for `highScore`. The method `update` picks a random integer between 1 and 11, inclusive. If the integer is 1 or 2 then the total goes to 0. Otherwise, the integer is added to the running total. Lastly, if there is a new high score, update it.

Open Unit3\Lab05\Panel05.java. Complete the Listener. What should the button do?

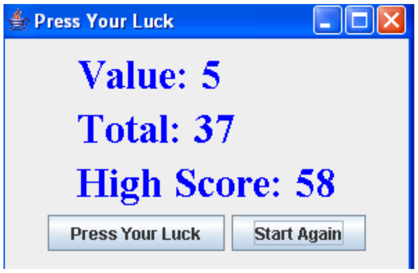
Open Unit3\Lab05\Driver05.java.

***Sample Run*** `Unit3Lab05.jar`

***Extension***
Create a second button that acts as an on-off switch. Include a javax.swing.timer that calls `update` every 500 ms. The text on the second button changes.



Value: 5
Total: 19
High Score: 58



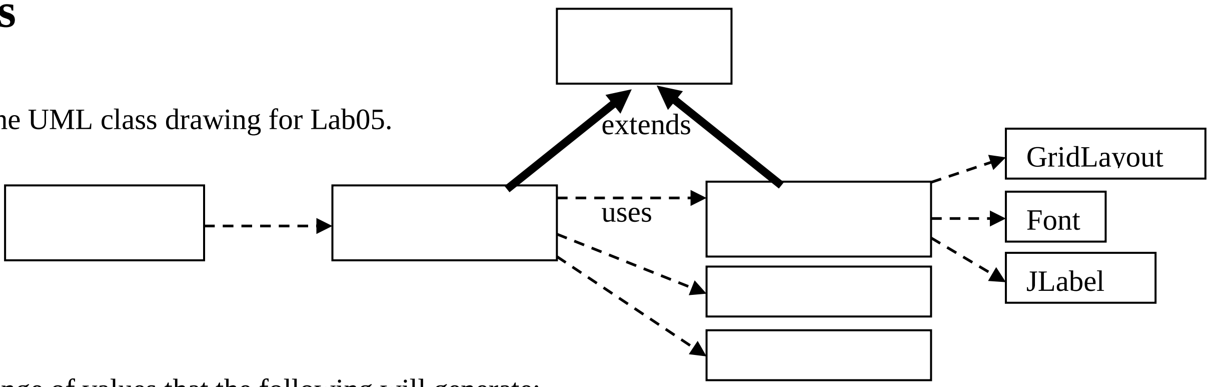
Value: 5
Total: 37
High Score: 58

***Sample Run*** `Unit3Lab05ext.jar`

# Exercises

## Lab05

1) Complete the UML class drawing for Lab05.



2) Give the range of values that the following will generate:

   a) `double x = Math.random();` ______________________

   b) `double x = Math.random() * 10;` ______________________

   c) `int n = (int)(Math.random() * 10);` ______________________

   d) `int n = (int)(1 + Math.random() * 10);` _____________________

3) Assign *n* a random integer value from 1 to 100 inclusive.

4) Assign *n* a random integer value from 2 to 100 inclusive such that n must be an even number.

5) Assign *x* a random integer value from 40 to 50, inclusive.

6) Assign *y* a random integer value from 40 to 50 (inclusive) such that *y* does not equal the *x* from Question 5.

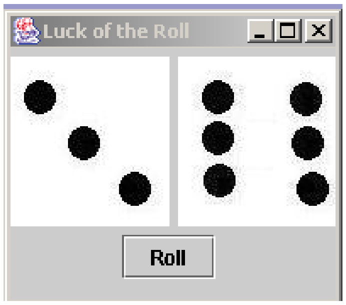
7) Assign *y* a different random integer value from 40 to 50 (inclusive) such that it does not equal its previous value.

8) Write the code to count the number of times a coin comes up heads in 10000 coin flips.

9) Write the code to count the number of times a six-sided die comes up 1 if you roll every day for a year.

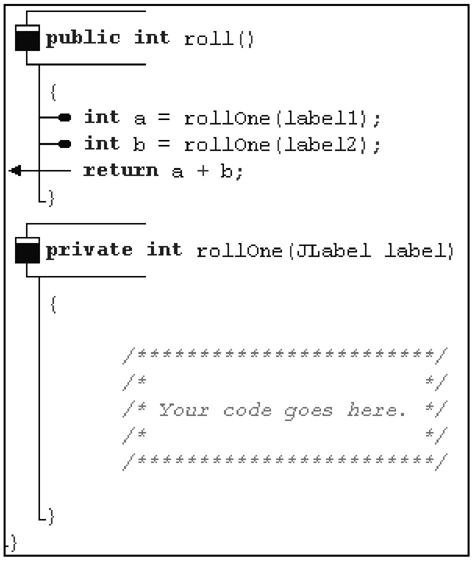
# Lab06
## Luck of the Roll

***Objective***
Images, switch statement, return statement.



***Background***
The panel has a Dice object and one button. The Dice object has two labels which display the dots. The Dice class also defines the method `roll`. The `roll` method rolls each die separately, i.e., it *calls* `rollOne` twice, and *returns* the sum of the dice.

`return` statements return values to the place where the method was called. Since the method `roll` was called from the panel, the value a+b will be returned to the panel. Similarly, method `rollOne` will return its value to where it was called from the method `roll`.



```
public int roll()
{
   int a = rollOne(label1);
   int b = rollOne(label2);
   return a + b;
}
private int rollOne(JLabel label)
{

      /************************/
      /*                      */
      /* Your code goes here. */
      /*                      */
      /************************/

}
}
```

Both these methods are *return methods*. You can recognize return method from two clues: because the code says **`int`** `roll` and **`int`** `rollOne`, and 2) because each has one (or more) **`return`** statements, usually placed at the end of the method.

The `rollOne` method generates a random integer {1,2,3,4,5,6} to model a roll of one die. It also sets the dots, which are ImageIcons. Rolling a 1 has to set “one.jpg,” rolling a 2 has to set “two.jpg,” etc. We could use an if-else ladder, but we can also use a **`switch`** statement to do this. A switch statement makes decisions much like an if-else ladder. Look at the Exercise #2 on the next page to see the syntax for a switch statement.

Whether you use `switch` or `if`, recall the image commands from Unit2 to put a .jpg on a graphics object:

```
   ImageIcon thomas = new ImageIcon("tj.jpg");
   g.drawImage(thomas.getImage(), 150, 75, 110, 150, null);
```

The Unit3 commands to put a .jpg on a label is somewhat simpler:

```
   ImageIcon thomas = new ImageIcon("tj.jpg");
   label.setIcon(thomas);
```

***Specification***
Open Unit3\Lab06\Dice.java. Implement the method `rollOne`. Pick a random number from one to six to simulate rolling a six-sided die. A label is passed as an argument to `rollOne`. Put an appropriate image on this label based on the value of the roll. Your method `rollOne` must have a return statement, or the compiler will complain "missing return statement."

Open Unit3\Lab06\Panel06.java. Write `actionPerformed`. Ask yourself: when the button is clicked, what should happen? Only if the roll is a seven should you display “Winner!” on the label.

Create Unit3\Lab06\Driver06.java. Create an appropriate driver.

***Sample Run*** `Unit3Lab06.jar`

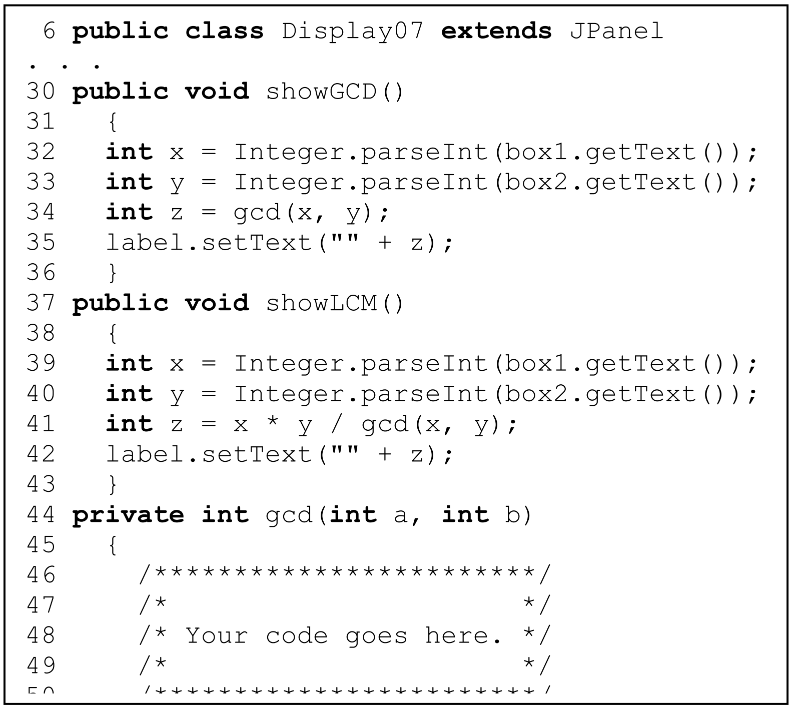
# Exercises

## Lab06

*Write the output exactly as it would appear on the screen. Check your prediction by typing each fragment into a driver program.*

<table>
<tr><th>#</th><th>Code Fragment</th><th>Output</th></tr>
<tr><td>1</td><td>for(int k=1; k<=5; k++)<br>  if(k==1 || k==3)<br>      System.out.println(“Odd.”);<br>  else if(k==2 || k==4)<br>      System.out.println(“Even.”);<br>  else<br>      System.out.println(“Foo.”);</td><td></td></tr>
<tr><td>2</td><td>for(int k=1; k<=5; k++)<br>switch(k)<br>  {<br>    case 1:<br>    case 3: System.out.println(“Odd.”);<br>      break;<br>    case 2:<br>    case 4: System.out.println(“Even.”);<br>      break;<br>    default: System.out.println(“Foo.”);<br>  }</td><td></td></tr>
<tr><td>3</td><td>int step = (int)(Math.random() * 3);<br>switch(step)<br>  {<br>   case 0: karel.move();<br>         break;<br>   case 1: karel.turnLeft();<br>         break;<br>   case 2: karel.turnRight();<br>           karel.move();<br>   }</td><td></td></tr>
<tr><td>4</td><td>for(int k=1; k<=5; k++)<br>  if(k == 1 || k % 2 == 0)<br>    System.out.println(“Hello.”);<br>  else<br>    System.out.println(“Good-bye.”);</td><td></td></tr>
<tr><td>5</td><td>for(int k=1; k<=5; k++)<br>  if(k == 1 && k % 2 == 0)<br>    System.out.println(“Hi.”);<br>  else<br>    System.out.println(“Bye.”);</td><td></td></tr>
<tr><td>6</td><td>public boolean isPrime(int a)<br>{<br>for(int k=a/2; k >= 2; k--)<br>   if(a % k == 0)<br>      return false;<br>return true;<br>}<br>System.out.println( isPrime(23) );<br>System.out.println( isPrime(25) );</td><td></td></tr>
</table>

# Discussion
## Helper Method



```
 6 public class Display07 extends JPanel
. . .
30 public void showGCD()
31   {
32   int x = Integer.parseInt(box1.getText());
33   int y = Integer.parseInt(box2.getText());
34   int z = gcd(x, y);
35   label.setText("" + z);
36   }
37 public void showLCM()
38   {
39   int x = Integer.parseInt(box1.getText());
40   int y = Integer.parseInt(box2.getText());
41   int z = x * y / gcd(x, y);
42   label.setText("" + z);
43   }
44 private int gcd(int a, int b)
45   {
46     /************************/
47     /*                      */
48     /* Your code goes here. */
49     /*                      */
50     /************************/
```

Study these instance methods in Display07. Each of these methods consists of *input*, *process*, and *output*. Input is taken from text boxes. Since the input from a text box is always a string, it must be parsed into an integer. A calculation is made (that's the process) and the result is displayed to a label.

Notice that each method calls a private helper method named `gcd` (lines 34 and 41) that actually does the work. In other words, the programmer has encapsulated the `gcd` calculation. Programmers split their programs into parts because it is easier to think and write that way. Also, if we ever came up with a faster way of finding the `gcd` we would only have to rewrite our algorithm once in the private `gcd` helper method.

The `gcd` helper method is private to the Display07 class. This method is tagged `private` in order to restrict access to it. Any outside object (in this case, the Panel07 object) is allowed to call the Display07's public methods `showGCD` and `showLCM`, but not its private method `gcd`. An API lists only a class's public methods.

Make sure you include a **return** statement when you write the method `gcd`. Look at the header for `gcd` (it's Line 44). Notice that in place of the keyword **`void`** is the keyword **`int`**. This is a signal that the method calculates and *returns* an integer value. That integer value is passed back to the place where `gcd` was called (the right-hand side of Line 34). Code for any non-void method will not compile without a return statement. All non-void methods require you to `return` a value of the specified type.

Here is some sample input and output. Study it to make sure you understand GCD and LCM.

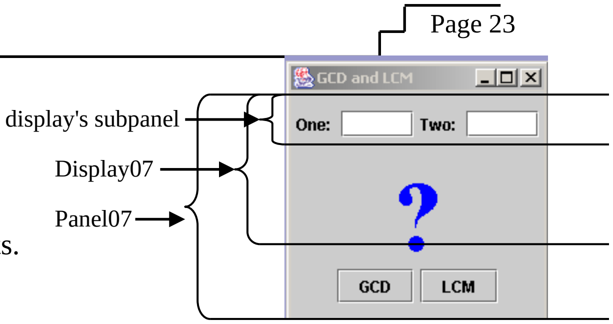
| a | b | GCD(a,b) | LCM(a,b) |
|---|---|---|---|
| 8 | 12 | 4 | 24 |
| 12 | 8 | 4 | 24 |
| 4 | 12 | 4 | 12 |

| a | b | GCD(a,b) | LCM(a,b) |
|---|---|---|---|
| 11 | 11 | 11 | 11 |
| 1 | 5 | 1 | 5 |
| 5 | 7 | 1 | 35 |

There are at least six algorithms to calculate the GCD of two numbers. One popular algorithm works by trial-and-error, as follows: test to see if some number "goes evenly into" both the given numbers. Systematically try different numbers until you find one which "goes evenly into" both the given numbers. What is the GCD if no number works?

Hint: see #6 on the previous page for code that systematically tries different numbers until it finds one that "goes evenly into" a single given number.

# Lab07
## GCD and LCM



### *Objective*
Local panels; using `return` statements to communicate results.

### *Background*
The panel "hasa" display and two buttons. The display "hasa" local subpanel to keep its four GUI components side by side. The local subpanel forces our prompt labels ("One: " and "Two: ") and our text boxes to remain together. These four components do not flow because they are all contained by the object named "panel."

Line 6: Display07 isa JPanel.

Lines 8-9: Declare 3 empty references in the fields.

Line 13: If you want, look up the Dimension class in the Java API.

Lines 15-25: instantiate, format, and add a JPanel referenced by `subpanel`.

Lines 17, 20, 21, and 24: add two JLabels and two JTextFields to the local subpanel.

Lines 19 and 23: you may look up `setHorizontalAlignment` which changes the justification of a textField. `SwingConstants` is an interface, but it does not specify any abstract methods. Instead, it defines useful constants as `public static final` variables which, by convention, are written in ALL CAPS.

```
 6 public class Display07 extends JPanel
 7  {
 8   private JLabel label;
 9   private JTextField box1, box2;
10   public Display07()
11    {
12    setLayout(new FlowLayout());
13    setPreferredSize(new Dimension(200, 125));
14 
15    JPanel subpanel = new JPanel();
16    subpanel.setLayout(new FlowLayout());
17    subpanel.add(new JLabel("One: "));
18    box1 = new JTextField("", 5);
19    box1.setHorizontalAlignment(SwingConstants.CENTER);
20    subpanel.add(box1);
21    subpanel.add(new JLabel("Two: "));
22    box2 = new JTextField("", 5);
23    box2.setHorizontalAlignment(SwingConstants.CENTER);
24    subpanel.add(box2);
25    add(subpanel);
26 
27    label = new JLabel("?");
28    label.setFont(new Font("Serif", Font.BOLD, 75));
29    label.setForeground(Color.blue);
30    add(label);
31    }
32  }
33  // instance methods showGCD(), showLCM(), gcd()
```

Lines 27-30: instantiate and add a JLabel for the "?" directly to Display07.

### *Specification*
Open Unit3\Lab07\Display07.java. Implement the private helper method `gcd`. One algorithm works by trial-and-error, as follows: test to see if some number “goes evenly into” both the given numbers. Systematically try different numbers until you find one which “goes evenly into” both the given numbers. If you use a different algorithm, such as Euclid's Algorithm, be prepared to show your teacher how it works.

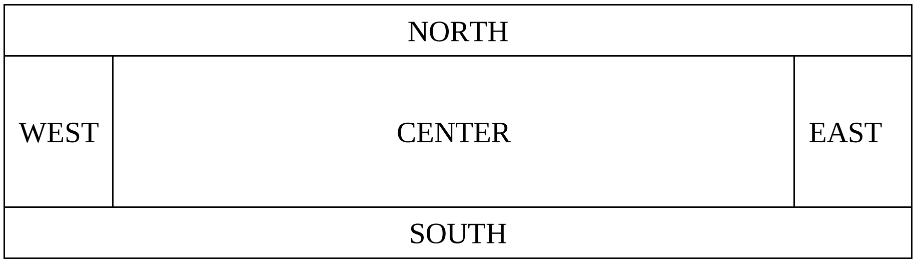
Create Unit3\Lab07\Panel07.java. In Panel07, instantiate and add a `display` object. Instantiate two buttons and add them to Panel07. Have the listeners of the buttons call the methods in `display`.

Create Unit3\Lab07\Driver07.java. Create an appropriate driver.

***Sample Run*** `Unit3Lab07.jar`

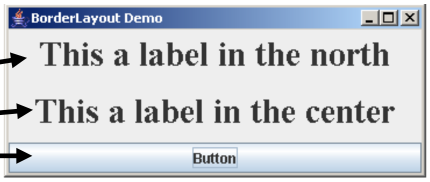
# Discussion
## Border and Grid Layouts

FlowLayout, BorderLayout, and GridLayout are the three kinds of LayoutManagers that we will use on panels. As you have seen, in FlowLayout the GUI components change position as you resize the panel. In contrast, the BorderLayout assigns each GUI component to one of five distinct regions as shown.



Conveniently, BorderLayout's `add` method takes two arguments which specify the GUI component and the region. Take care to add only one component per region. If you add more than one component to a particular region then only the last component will show.

```
setLayout(new BorderLayout());
...
add(label1, BorderLayout.NORTH);
add(label2, BorderLayout.CENTER);
add(button, BorderLayout.SOUTH);
```



This a label in the north

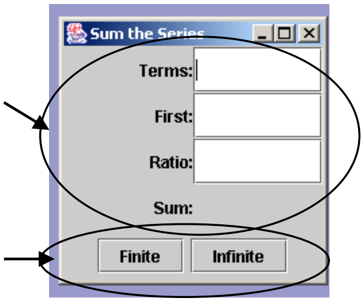
This a label in the center

Be careful! If you don't specify a region, the default region is CENTER. A common error is to not specify a region and then ask the teacher why only the last object shows up, covering everything. The teacher will answer that you have added components on top of each other in the center.

You do not have to use all the regions. Any region that doesn't get a component will disappear. Often we will not use the east or west regions, in which case the center region will expand horizontally.

Panel08 has a BorderLayout that uses the CENTER and the SOUTH. Panel08's center region hasa Display08, which hasa GridLayout. GridLayout makes rows and columns.

```
public Display08()
 {
   setLayout(new GridLayout(4, 2));
   add(new JLabel("Terms:", SwingConstants.RIGHT));
   box1 = new JTextField("", 5);
   add(box1);
   // 6 more components
```



panel hasa display in CENTER.
display hasa 4x2 GridLayout.

panel hasa subpanel in SOUTH.
subpanel hasa 1x2 GridLayout.

Notice that Display08 has four rows and two columns in its grid. Labels and/or textfields are added from left to right across each row of the grid from the top of the grid to the bottom. Be sure to add exactly the number of components that the grid is expecting (in this case, 8), or unpredictable things may happen. Add anonymous blank labels if you want cells of the grid to appear unoccupied.

Notice that Panel08's south region hasa subpanel, which hasa Grid Layout with two buttons. See Lab07.

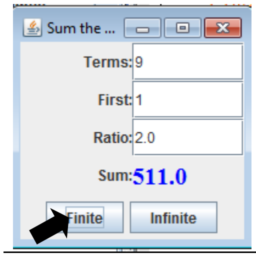
Convention dictates rows first, then columns. You can use RC Cola (stands for Royal Crown Cola) as a mnemonic to remember that we always treat the rows (R) first and the columns (C) second.

# Lab08
## Sum of a Series

***Objective***
Work with GridLayouts in CENTER and SOUTH. Implement an algorithm and a formula.

***Background***
A geometric series is the sum of a sequence of terms, where each term is generated by a ratio. A series may be finite or infinite. All finite series have sums. Some infinite series have sums, depending on whether the absolute value of the ratio is strictly less than one; if not, then the series diverges and does not have a sum.

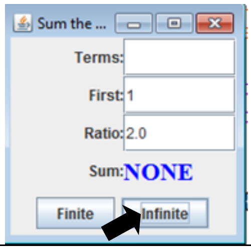
Study the four series shown below. The two on the left are finite and both have sums. The two on the right are infinite. The top infinite series diverges and does not have a sum; the bottom infinite series converges and has a sum.



Terms: 9
First: 1
Ratio: 2.0
Sum: 511.0

Finite, 9 terms:
1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 + 256

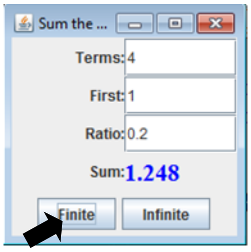
= 511.0



Terms:
First: 1
Ratio: 2.0
Sum: NONE

Infinite series diverges:

1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 + 256 + . . .
No sum

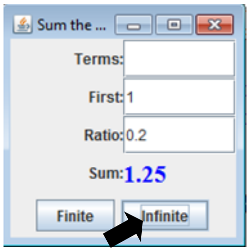


Terms: 4
First: 1
Ratio: 0.2
Sum: 1.248

Finite, 4 terms:

1 + 0.2 + 0.04 + 0.008

= 1.248



Terms:
First: 1
Ratio: 0.2
Sum: 1.25

Infinite series converges:

1 + 0.2 + 0.04 + 0.008 + 0.0016 + 0.00032 + . . .

= 1.25

What primitive type is *terms*? ____What primitive type is *first*? ______What primitive type is *ratio*? _________

We will use a for-loop to calculate the sum of a finite series. Inside the for-loop, calculate one term from the previous term by multiplying by the common ratio. Given the first term, you can find the second term. From the second term you can find the third, from the third the fourth, and so on. Keep a running total (the sum) of the terms as you go. At the end of the loop, you will have calculated the sum of the entire series.

The sum of an infinite series (if it has a sum) can be found with the formula $\frac{first}{1-ratio}$.

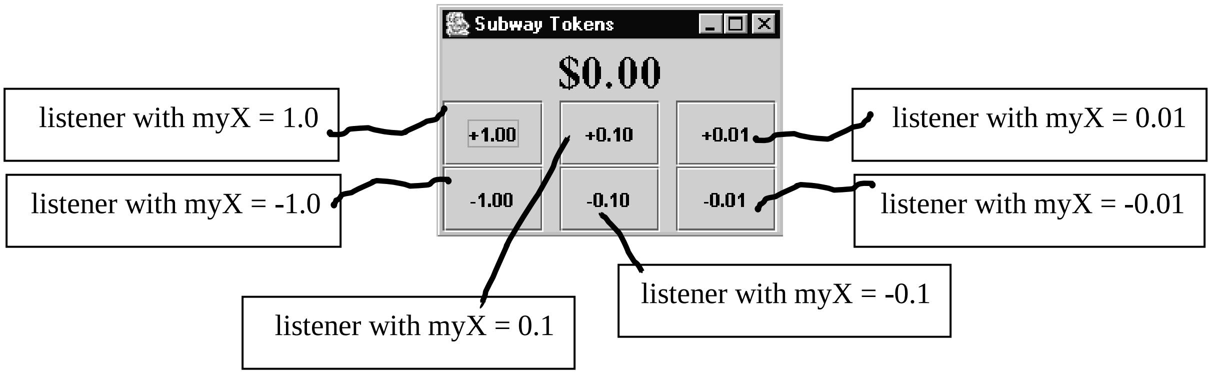
***Specification***
Open Unit3\Lab08\Display08.java. Implement the methods `sumFinite` and `sumInfinite` using the algorithms above. All finite series have a sum. If the infinite series has no sum, display "NONE."

Open Unit3\Lab08\Panel08.java. Put the Display08 object in the center and two buttons side-by-side in a subpanel in the south. Make the buttons call the Display08's methods.

Open Unit3\Lab08\Driver08.java. Compile and run.

***Sample Run***
`Unit3Lab08.jar`

# Discussion
## Generic Listeners



Looking at Lab09's center panel, we see six buttons and six listeners, or 12 different objects. Do we need to write 12 different classes, one for each object? No! We can write one button-making method and call it six different times. The listener class needs a constructor with an argument, which gets stored in a private field. When we call the listener's constructor, we pass an argument, either 1.0, -1.0, 0.1, etc., so that it creates a unique object. (Previous listener classes used the default constructor and had no fields.) The code shows how to make a *generic listener*.

```
16  total = 0.0;
. . .
39 private class Listener implements ActionListener
40 {
41   private double myX;
42   public Listener(double x)
43   {
44      myX = x;
45   }
46   public void actionPerformed(ActionEvent e)
47   {
48
49
50   }
51 }
```

Line 41: the field `myX` stores doubles. It will store a different double value for each listener object.

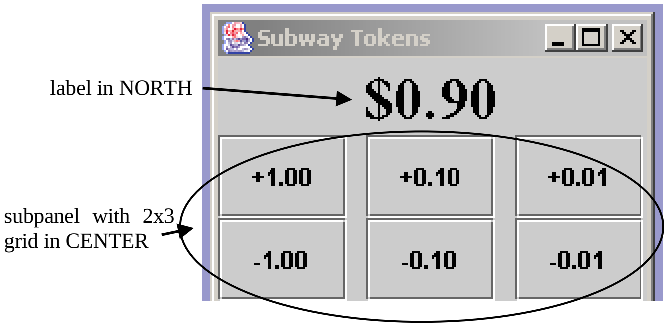
Lines 42-45 is the listener's constructor, which requires a double type argument. It takes that value and stores it in `myX` (Line 44). For example, when we instantiate the Listener, we might write the code **`new`** `Listener(0.10);`. After the code is finished executing, that listener will have a `myX` that stores 0.1. We will eventually instantiate six button objects with six listener objects, each with a different value in `myX`.

When the user clicks the button, that button's `myX` is added to `total` and displayed on the label with a "$" symbol. Write that code on lines 48 and 49 in `actionPerformed` above.

If you typed in that code as described, the label would sometimes show lots of decimal places and sometimes not. That is unwelcome behavior when the number represents dollars and cents. Java's solution is typically Javanese: first, import a class, namely `java.text.DecimalFormat` . Instantiate a DecimalFormat object, whose 1-arg constructor takes a String "$0.00" that becomes the formatting pattern. DecimalFormat has several `format` methods (inherited from NumberFormat), one of which accepts a `double` argument and returns that argument as the formatted string. You know that labels have a `setText` method that accepts a string argument. Now put it all together, to make the label display `total` formatted as dollars and cents.

```
46 public void actionPerformed(ActionEvent e)
47  {
48
49
50
51  }
```

# Lab09
## Subway Tokens



***Objective***
Create a generic factory method for buttons.

***Background***
Panel09 has a BorderLayout (Line 15), with a label in the north and a grid in the center sub-panel.

```
13 public Panel09()
14  {
15  setLayout(new BorderLayout());
16  total = 0.0;
17
18  JPanel panel = new JPanel();
19  panel.setLayout(new GridLayout(2,3,10,0));
20  add(panel, BorderLayout.CENTER);
21  addButton(panel, "+1.00", 1.0);
22  addButton(panel, "+0.10", 0.1);
23  addButton(panel, "+0.01", 0.01);
24  addButton(panel, "-1.00", -1.0);
25  addButton(panel, "-0.10", -0.1);
26  addButton(panel, "-0.01", -0.01);
```

Line 16. The `total` variable is set to zero. What does `total` do?

Lines 18-20. Instantiate, format, and add a local JPanel. GridLayout can take 4 arguments. Notice the 2 rows and 3 columns. The 10 puts 10 pixels between the columns and the 0 puts 0 pixels between the rows.

Lines 21-26. Add six anonymous button objects to the panel by calling `addButton`. Notice that the method `addButton` requires three arguments. What type is each argument? ___________, __________, ___________. Each argument is going to be used in the `addButton` method. What is the purpose of each argument?

What are the three steps in creating a button? (See page Three-4.) The `addButton` method must instantiate a button, passing the string. The button must register a listener, passing the double value to the listener constructor. The panel must add the button. Write the private *factory method* `addButton` below:

```
33  private void addButton(JPanel panel, String s, double x)
34   {
35
36
37
38   }
```

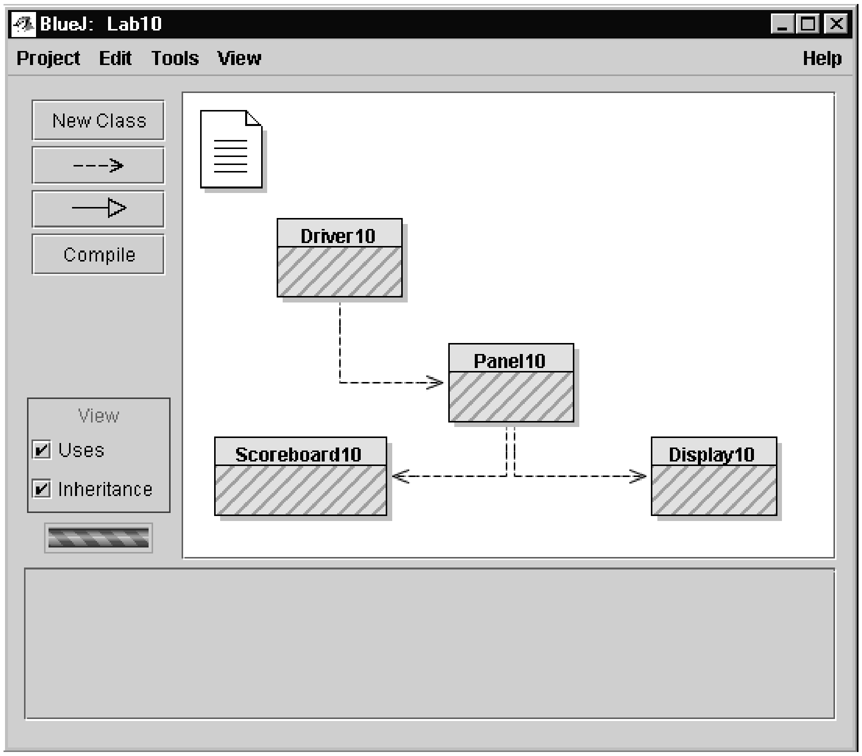
***Specification***
Open Unit3\Lab09\Panel09.java. Complete the implementation of the two methods `addButton` and `actionPerformed`. The `actionPerformed` method must adjust the total appropriately, format the total as currency, and update the display.

Create Unit3\Lab09\Driver09.java. Create an appropriate driver.

***Sample Run***
`Unit3Lab09.jar`

# Discussion
## Scoreboard

Here is an object diagram produced by the program BlueJ:



The dotted lines indicate a *hasa* (or *uses)* relationship. Driver10 *hasa* Panel10, as usual. Panel10 in turn *hasa* Display10 and a Scoreboard10. Notice that Display10 and Scoreboard10 are not related to each other. In fact, the scoreboard and the display do not interact at all. The objects are like black boxes, in that all methods and private data are *encapsulated* within each object. All communication of data is accomplished through methods' arguments and return statements. Furthermore, the communication between scoreboard and display is managed by a third object, the panel.

Two methods in Display10 (`guessHigh` and `pickNext`) work together to pick random numbers from 1 to 9 and compare each new number to its immediate predecessor. Then `guessHigh` returns a boolean, communicating whether or not the new number is higher or lower than the previous number.

A certain method in Scoreboard10 (called `update`) accepts a boolean through its arguments. Depending on the value of that boolean, the `run` and the `max` may be updated, and some labels set to display the results.

The panel's job is to manage the scoreboard, the display, the two buttons, and their listeners. The panel passes information from the display to the scoreboard by a listener with code like this:

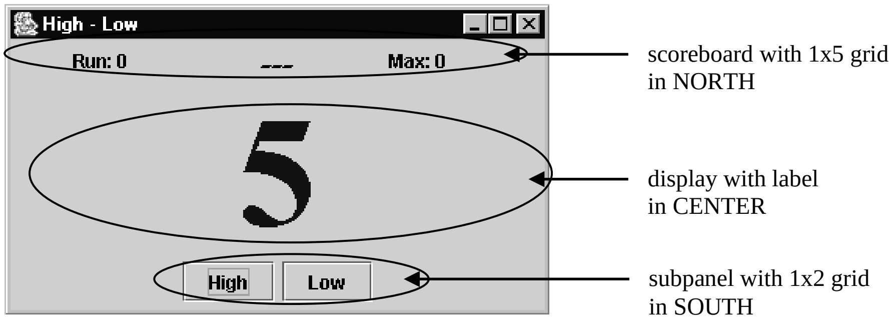
```
scoreboard.update(display.guessHigh());
```

Explain in your own words how the code above works:

# Lab10
## High - Low

***Objective***
Using multiple classes and methods that pass arguments and return values.

***Background***
The panel *hasa* BorderLayout holding the scoreboard, the display, the buttons, and their listeners.



scoreboard with 1x5 grid
in NORTH

display with label
in CENTER

subpanel with 1x2 grid
in SOUTH

The scoreboard is added to the north region of the panel. The display is added to the center region of the panel. A separate panel defined locally in the Panel10 constructor takes care of the buttons in the south. When you press the High button, this code is in its ActionListener:

```
scoreboard.update(display.guessHigh());
```

```
30  public boolean guessHigh()
31    {
32       pickNext();
33       return next > last;
34    }
```

Here is the code for `guessHigh` in the Display07 object. Line 32 calls `pickNext`. The method `pickNext` stores the old number and picks a new integer between 1 and 9 inclusive. `pickNext` is written to make sure that it does not pick the same number twice in a row. After `pickNext` is finished, the code executes line 33, which returns a boolean `true` if the next number is greater than the previous number, and `false` if it is less. The method `guessLow` also calls `pickNext`, but its return statement is **`return`** `next < last;`. Be able to explain why that makes sense.

***Specification***
Open Unit3\Lab10\Display.java. Implement the method `pickNext` to store the old number in `last` and pick a new integer `next` between 1 and 9 inclusive. Write the code so that `pickNext` generates a random number again if it happened to pick the same number as it did before.

Open Unit3\Lab10\Scoreboard10.java. Implement the method `update`. The `update` method requires one argument of type boolean. This argument indicates whether or not the user's most recent guess was correct. If the argument is true, set the scoreboard's label to "Yes", update the current run of correct guesses, and update the maximum streak of correct guesses. If false, set the label to "—No—", and set the current run of correct guesses to zero.

Open Unit3\Lab10\Panel10.java. Compile.

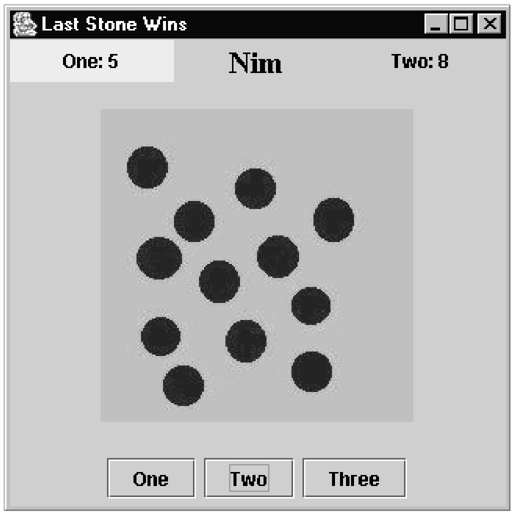
Create Unit3\Lab10\Driver10.java. Create an appropriate driver.

***Sample Run***
`Unit3Lab10.jar`

# Lab11
## Last Stone Wins

***Objective***
Making a simple game with buttons and images.

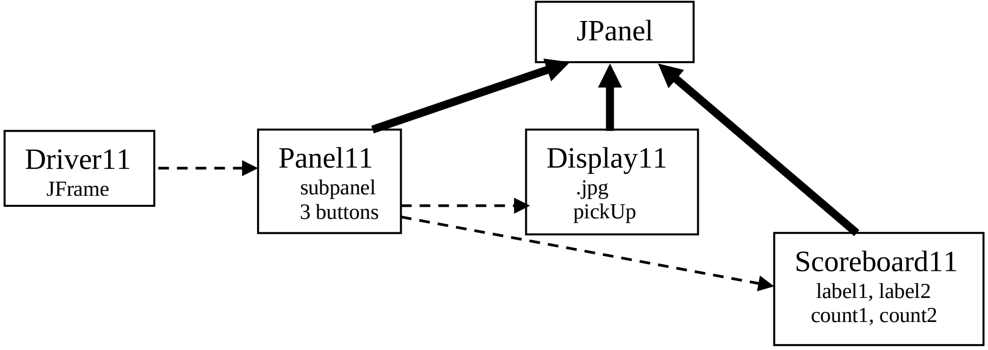
***Background***
This version of Nim uses multiple classes as in Lab10: driver, panel, display, and perhaps a scoreboard. It uses the `addButton` and listener field ideas from Lab09. It uses the image icon concepts from Lab06. It uses grid layouts and border layouts as appropriate. Your goal in planning a solution is to make your work as simple and as understandable as possible.



The game you're writing is Nim for two people. Our Nim has one pile with 12 stones from which players alternate removing stones. Each player may remove 1, 2, or 3 stones per turn. You must take at least one stone on each turn. Whoever takes the last stone wins.

Hint: the program doesn't actually pick up 2 stones. Instead, it subtracts 2 from the amount of stones, and displays an image with the correct number of stones.

Think about the responsibilities of each class:



In Tic Tac Toe, if both players play perfectly, the result is always a tie. In Nim, the second player has a strategy so that he or she always wins. What is that strategy?

***Specification***
The .jpg files provided for you are called `Unit3\Lab11\stones1.jpg`, `\stones2.jpg`, and so on.

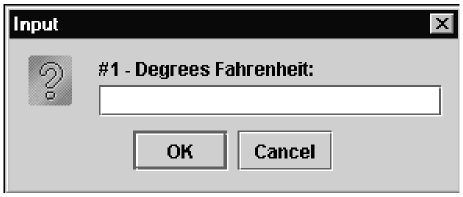
You create all the other files. Make a panel and a display as in Lab10. For the three buttons, make one listener, as in Lab09.

***Extension***
Add a scoreboard to the panel that keeps track of each person's number of wins. The scoreboard also changes its background color to indicate who is playing at the moment.

***Sample Run***
`Unit3Lab11.jar`

# Discussion
## JOptionPane

Sometimes making a GUI from scratch is tedious. Luckily, Java has a JOptionPane class which contains static methods that produce standardized GUIs. Without you yourself creating a frame or a panel or a button or any other type of object, the method `showInputDialog` produces a small GUI that allows the user to input data. All data is read in as a string, just like reading text from a text box in Unit 3.



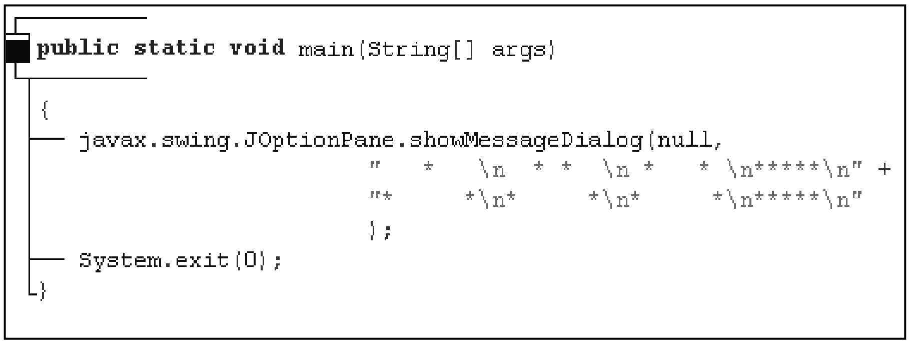
Because all data is read in as a string, numeric data must be converted from text to numbers with either `parseInt` or `parseDouble`. Of course, if the data is words then no parsing is required. For the dialog box shown above, one item of data could be prompted, read, parsed, and stored into a variable `fahrenheit` with the commands:

```
fahrenheit = Double.parseDouble(
                JOptionPane.showInputDialog("#1 - Degrees Fahrenheit:"));
```

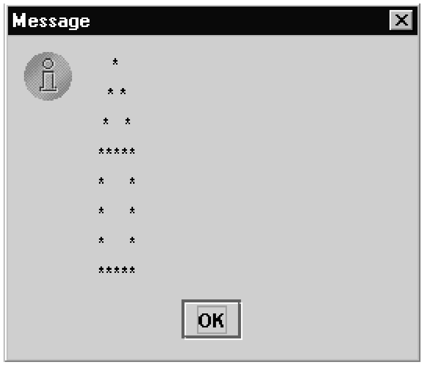
Where is the prompt?________________________ Which command returns a string?_________________

Which command parses the string? _____________ Which command stores the number? _________________

There is also a JOptionPane class method `showMessageDialog` that can be used to output text to a GUI.



```
public static void main(String[] args)
{
   javax.swing.JOptionPane.showMessageDialog(null,
                        "  *   \n * *  \n *   * \n*****\n" +
                        "*     *\n*     *\n*     *\n*****\n"
                        );
   System.exit(0);
}
```



```
  *
 * *
*   *
*****
*   *
*   *
*   *
*****
```

Be careful! Don’t forget to include the first argument, `null`, when calling `showMessageDialog`.

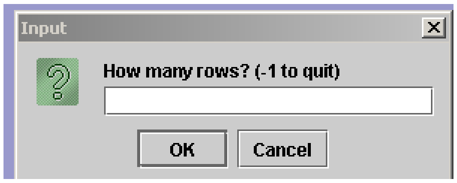
Notice that you can display output on more than one line using the `"\n"` escape sequence for a newline.

The last line in an application that uses JOptionPane should always be `System.exit(0),` in order to close the GUI thread that is running behind the scene.

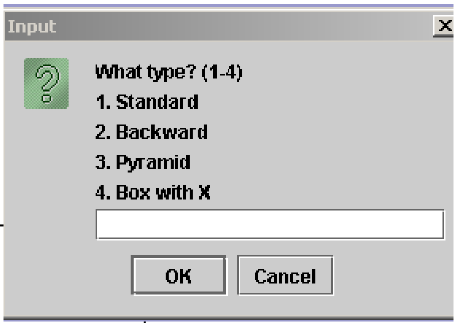
# Discussion
## I/O Window output

The next three labs will prompt the user to enter data, process it, and output the results to the default I/O window.

The user will enter data through Java's built-in JOptionPane methods. The JOptionPane class provides standard GUI pop-up panes, like the ones pictured at the right. You don't always have to build your own GUIs from scratch.



How many rows? (-1 to quit)



What type? (1-4)
1. Standard
2. Backward
3. Pyramid
4. Box with X

Here is Driver12’s main method:

```
 4 import javax.swing.JOptionPane;
 5 public class Driver12
 6 {
 7   public static void main(String[] args)
 8   {
 9     while(true)
10     {
11       int rows = Integer.parseInt(
12                   JOptionPane.showInputDialog(
13                       "How many rows? (-1 to quit)"));
16       if(rows == -1)
17       {
18          System.out.println("Bye-bye!");
19          System.exit(0);
20       }
21       String message = "What type? (1-4)";
22       message = message + "\n1. Standard";
23       message = message + "\n2. Backward";
24       message = message + "\n3. Pyramid";
25       message = message + "\n4. Box with X";
26       int type = Integer.parseInt(
                   JOptionPane.showInputDialog(message));
27       switch(type)
28       {
29          case 1: standard(rows);
30             break;
31          case 2: backward(rows);
32             break;
33          case 3: pyramid(rows);
34             break;
35          case 4: box(rows);
36             break;
37          default: System.out.println("Not valid type.");
38       }
39       System.out.println();
40     }
41  }
42  public static void standard(int n)
43    {
```

Line 9: `while(true)` looks like an infinite loop. However, if the user enters a -1, then Lines 16-19 will end the program.

Line 11: since `showInputDialog` returns a string, we must parse the result. We store it in *rows*.

Line 12: `showInputDialog` is a class method in the JOptionPane class. Line 13 is the *prompt string*. Lines 11-13 make the GUI window shown above.

Lines 21-25: this is a clever way to create a long string, later used to prompt the user. `"\n"` makes a new line in the string.

Line 26: creates the GUI window, waits for the user to click OK, parses the result, and stores it in `type`.

Lines 27-39: the `switch` statement calls the methods, passing the argument `rows`, that draws the patterns.

# Lab12
## Asterisks

***Objective***
Print patterns of asterisks using nested loops.

***Background***
The next three labs are not object oriented programming and do not output to a GUI window. Instead, each application outputs to a default I/O window. Currently, the only output from the Driver12.java shell is:

```
This type is not currently supported.
```

Your job is to make each method print the given pattern, as shown below. The integer argument *n* specifies the number of rows. Use an outer for-loop to control the number of rows. Inside that for-loop, use a combination of for-loops and if-statements to control the number of asterisks on each row. Use `print` or `println` to control the line returns. As a starting example, this code with two *nested loops* prints a square of asterisks arranged in *n* rows and *n* columns. If *n* = 4, you get the square pattern shown at the right.

```
for (int r = 1; r <= n; r++)
   {
     for (int c = 1; c <= n; c++)
          System.out.print("*");
     System.out.println();
   }
```

```
****
****
****
****
```

| # | Type | *n* equals 7 | # | Type | *n* equals 7 |
|---|---|---|---|---|---|
| **1** | Standard | *<br>**<br>***<br>****<br>*****<br>******<br>******* | **4** | Box with X | *******<br>**   **<br>* * * *<br>*  *  *<br>* * * *<br>**   **<br>******* |
| **2** | Backward |       *<br>     **<br>    ***<br>   ****<br>  *****<br> ******<br>******* | | | |
| **3** | Pyramid is a Backward plus a Standard |       *<br>     ***<br>    *****<br>   *******<br>  *********<br> ***********<br>************* | | | |

For the Box with X, start by understanding the output of these nested for-loops:

```
for (int row = 1; row <= n; row++)
  {
   for (int col = 1; col <= n; col++)
     if( (row + col) % 2 == 0 )    //if its true
       System.out.print("*");
     else
       System.out.print(“ ”);
   System.out.println();
  }
```

***Specification***
Open Unit3\Lab12\Driver12.java. Implement each of the asterisk pattern methods.

***Sample Run***
`Unit3Lab12.jar`

# Lab13
## The Necklace

***Objective***
Prompt the user for data. Error-check the data. Implement the "necklace" algorithm.

***Background***
A digit is one of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. A necklace begins with any two of these digits. Prompt the user to enter two digits, `first` and `second`. Suppose the user enters 4 and 7. Okay, print `4, 7`.

`do:`

- Add 4 and 7 to get 11. Extract the one's digit from 11. Print it, `1`.
- Add 7 and 1 to get 8. Extract the one's digit from 8. Print it, `8`.
- Add 1 and 8 to get 9. Extract the one's digit from 9. Print it, `9`.
- Add 8 and 9 to get 17. Extract the one's digit from 17. Print it, `7`.
- Continue this "necklace" pattern. Three variables, `a`, `b`, and `c`, are all you need.

`while` the digits 4 and 7 do not re-appear in the necklace.

After that works, add a counter keep track of how many times you had to add to get back to `4,7`. In our example you will see:

```
4 7 1 8 9 7 6 3 9 2 1 3 4 7
Iterations: 12
```

There are a hundred possible ordered pairs of digits to start with. Do they all take twelve iterations to return to where they started? The answer is no. For 3 and 7 you will see:

```
3 7 0 7 7 4 1 5 6 1 7 8 5 3 8 1 9 0 9 9 8 7 5 2 7 9 6 5 1 6 7 3 0 3
3 6 9 5 4 9 3 2 5 7 2 9 1 0 1 1 2 3 5 8 3 1 4 5 9 4 3 7
Iterations: 60
```

***Specification***
Create filename Unit3\Lab13\Driver13.java. Prompt the user for two numbers. You will need to store permanent copies of `first` and `second`. You can do the rest of the lab with variables `a`, `b`, and `c`. Generate each necklace number, printing as you go. Count the number of times to get back to the starting numbers. Repeat until the user enters -1.

***Extension***
If the user enters a number that is not a single digit, prompt him or her repeatedly until a single digit is entered.

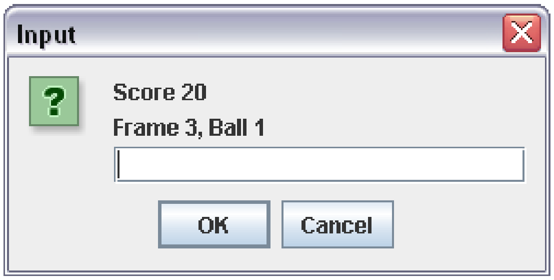
***Sample Run***
`Unit3Lab13.jar`

***Extension using Unit 2 graphics***
`GraphicsNecklace.jar`

# Lab13
## Exercises

| code | output |
|---|---|
| `for(int k=1; k<=5; k++)`<br>`System.out.print("" + k);` | |
| `for(int k=1; k<=5; k++)`<br>`System.out.println("" + k);` | |
| `System.out.print("1\n23\t4\");` | |

# Lab14
## Bowling



Score 20
Frame 3, Ball 1

***Objective***
Practice with user input, loops, and integers.

***Background***
Here is a not-very-precise algorithm to score a (simplified) game of bowling for one person.

1. A game of bowling takes ten frames. In each frame, the bowler throws one ball to knock down ten pins. If they aren't all knocked over, the bowler gets to throw a second ball.

2. Enter the number of pins that were knocked down by the first ball. Use a JOptionPane window, displaying the current score, frame, and ball, like the one shown above.

3. If the number entered is impossible (say, greater than 10), prompt the user to enter it again, repeatedly, if necessary. Otherwise, add the pins to the total score.

4. If all tens pins are knocked down by the first ball, the frame is over. Otherwise, prompt the user to enter the number of pins knocked down by the second ball. If the number entered is impossible, prompt to enter it again, repeatedly, if necessary. If the sum of the pins reported for the two balls is impossible, prompt to enter it again, repeatedly, if necessary. When the number is satisfactory, add the pins to the total score.

***Specification***
Create Unit3\Lab14\Bowling.java. Write the code to score a game of bowling for one person. You will need variables for `frame`, `ball`, `pins1`, `pins2`, and `score`. Display a welcoming message and a closing message. Do some error-checking. Use the `Unit3Lab14.jar` as a model.

***Sample Run***
`Unit3Lab14.jar`

***Extensions***
The real game of bowling rewards bowlers for spares and strikes. A *spare* occurs when all ten pins are knocked down using both balls in one frame. In that case, the pins knocked down on the next ball are counted twice, once in the frame before and once in its own frame. In the box score below, since frame #2 has a spare, the next ball in frame #3 is counted once in frame #2 and once for itself.

A *strike* occurs when all ten pins are knocked down on the first ball. In that case, no second ball is rolled for that frame and the pins knocked down with the next two balls are counted twice. *Each* strike counts the next two balls twice. Since frame #4 has a strike, the balls in frames #5 and #6 are each counted twice. Study the scores below, which is what you would see in a bowling alley. Finally, if a strike or spare is rolled in the tenth frame then an appropriate number of bonus balls are rolled. This scoring system works out so that a perfect game of 12 strikes earns a score of 300.

| frame | 1 | | 2 | | 3 | | 4 | | 5 | | 6 | | 7 | | 8 | | 9 | | 10 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pins | 5 | 0 | 5 | 5 | 3 | 0 | 10 | | 10 | | 10 | | 0 | 1 | | | | | | | |
| score | 5 | 5 | 10 | 18 | 21 | 21 | 51 | | 71 | | 82 | | 82 | 83 | | | | | | | |

You don't (yet) know about 2-dimensional arrays, but in this lab you can still calculate the total score that follows the real scoring rules. Just display the score at the end.

# Lab15
## More GUI components

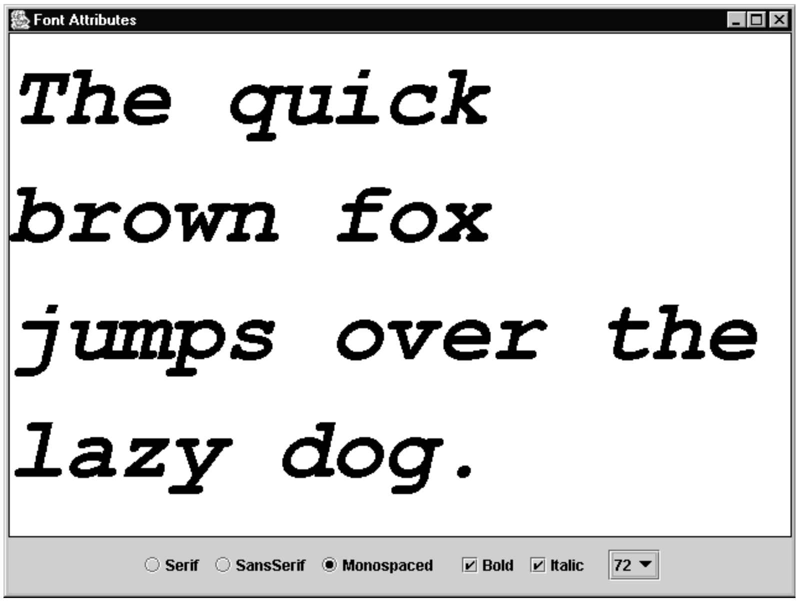
***Objective***
Use a text area, radio buttons, check boxes, a drop-down menu, and item listeners.

***Background***
Panel15 has a border layout. The center has a JTextArea with lineWrap turned on. The south is formatted in a flowLayout and has a NamePanel, a StylePanel, and a SizePanel.

A text area allows you to display multiple lines of text, which is not possible with a label or a text box.

The NamePanel has three objects of class JRadioButton. The StylePanel has two objects of class JCheckBox. The drop down menu in SizePanel is an object of the class JComboBox.

Each of these panels has factory constructors with one private field.



The quick brown fox jumps over the lazy dog.

○ Serif ○ SansSerif ● Monospaced ☑ Bold ☑ Italic 72 ▾

Each of these objects is similar to a button, but different. These objects register a listener using the command `addItemListener`. The Listener class implements the ItemListener interface, so that all ItemListeners must have a method named `itemStateChanged(ItemEvent e)`.

Three panels and the driver are given to you. You have to put them all together by coding Panel15.

***Specification***
Open Unit3\Lab15\Panel15.java. Set the layout and add the text area and a sub-panel. Add the three sub-sub-panels to the sub-panel in the south. By looking at the NamePanel constructor, observe that the text area is passed as an argument to each of the NamePanel, StylePanel, and SizePanel constructors. Passing a reference (or pointer) to the text area is how each sub-panel gains access to the JTextArea.

Open Unit3\Lab15\NamePanel.java. Note how the constructor takes an argument of type JTextArea. This is how NamePanel gains access to the JTextArea.

Open Unit3\Lab15\StylePanel.java. Note how the constructor takes an argument of type JTextArea. This is how StylePanel gains access to the JTextArea.

Open Unit3\Lab15\SizePanel.java. Note how the constructor takes an argument of type JTextArea. This is how SizePanel gains access to the JTextArea.

Open Unit3\Lab15\Driver15.java. Compile and run.

***Sample Run***
`Unit3Lab15.jar`

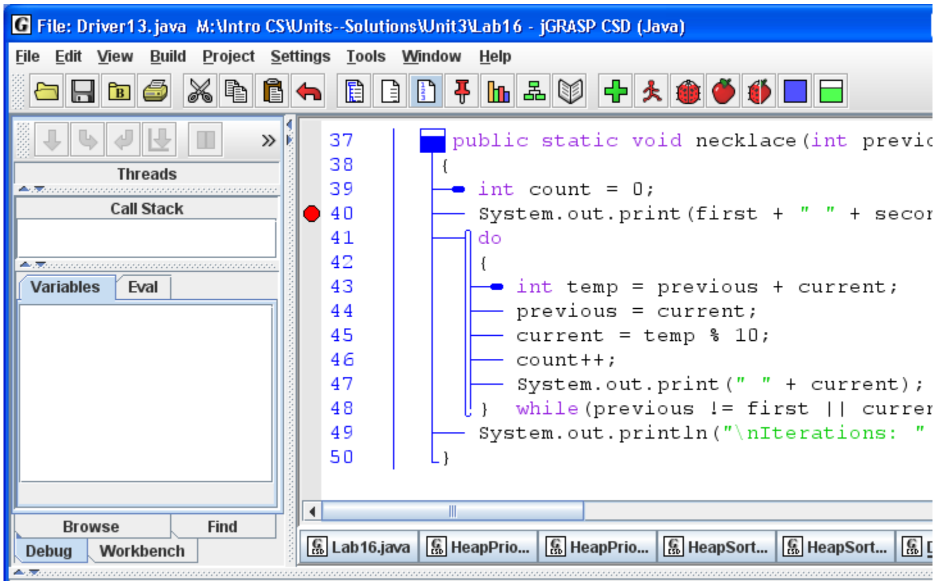
# Discussion
## The Debugger

Sometimes, even after our program compiles, we know there is a run-time error in our code. The JGrasp *debugger* allows us to step through the code line by line, so that we may stop at every step to see if the program is still working the way we want it to. This allows us to find the exact point at which our program goes wrong, so that the code becomes much easier to correct.

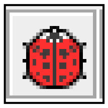
In this example, our program doesn't actually have any errors in it. But it is useful to run the debugger on it anyway, so that we can understand how it works and what information it can tell us.

### → How do I turn it on?

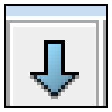
First, you must insert what we call a *breakpoint* somewhere in the code. The image below shows the breakpoints, like red stop signs, which have been inserted at line 40. These were inserted by left-clicking once in the gray stripe down the left margin next to each line.



```
37    public static void necklace(int previc
38    {
39       int count = 0;
40       System.out.print(first + " " + secor
41       do
42       {
43          int temp = previous + current;
44          previous = current;
45          current = temp % 10;
46          count++;
47          System.out.print(" " + current);
48       } while(previous != first || currer
49       System.out.println("\nIterations: "
50    }
```



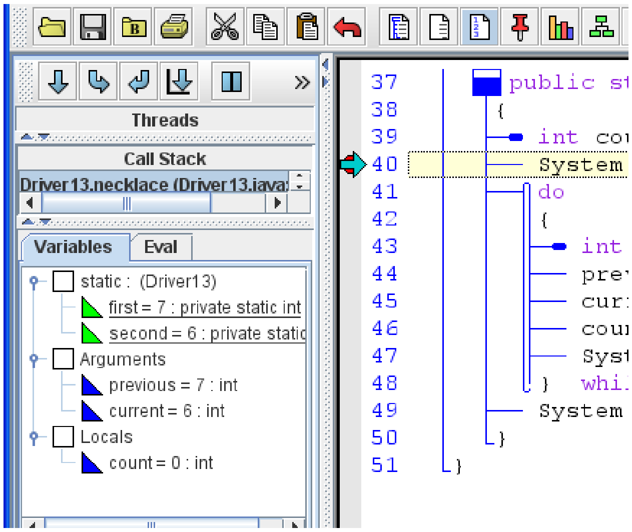
To actually start the debugger, click the ladybug icon to the right of the 'Run' icon.



The program will begin to run, but as soon as it reaches line 40, it will stop. Once you are ready to move on, you can hit the 'Step' button on the left menu, which will cause the debugger to continue executing the program line-by-line until the program is complete.

### → What am I looking at when it runs?

When you run the JGrasp Debugger, a window opens in the left pane that shows the *values* of all *variables* that have been defined to that point. At each breakpoint, you can check whether the variables you've defined are set to the values that you expect. As soon as you see an unexpected number, you can conclude that your code is doing something you don't want it to do. Fix it.



Driver13.necklace(Driver13.java:

static : (Driver13)
first = 7 : private static int
second = 6 : private static
Arguments
previous = 7 : int
current = 6 : int
Locals
count = 0 : int

```
37    public st
38    {
39       int co
40       System
41       do
42       {
43          int
44          pre
45          cur
46          cou
47          Syst
48       } whi.
49       System
50    }
51 }
```

### → How can I get it to move backward to the last breakpoint?

Bad news, you can't. The only thing you can do is stop it (by pushing the 'End' button) and restarting by pushing the ladybug button again.

# Lab16
## The jGrasp Debugger

This exercise requires you to have a working solution to Lab13, The Necklace.

> The user inputs any two digits between 0 and 9. The necklace begins with those two digits and each subsequent digit is obtained by adding the previous two digits and keeping only the ones digit of the sum. For example, if 7 and 6 are the selected digits, the next term in the necklace is 3, because 7 + 6 = 13, and the ones digit of 13 is 3. The term after that is 9, because now the previous two terms are 6 and 3. The necklace continues 7..6..3..9..2..1..3..4..7..1..8..9..7..6…at which point it stops because the final two terms are the same as the initial two terms. Twelve iterations were required to arrive at the same two digits as the first two.

**→If you have not finished Lab13, then a working program can be found in the Lab16 folder.**

Compile and run the program and notice that it actually works. Study the code and understand it.

**→Set a breakpoint at Line 40. Then click on the Ladybug button to start the debugger. As in the example above, enter 7 for the first digit and 6 for the second digit.**

**→Notice the variables in the left pane are grouped according to "static," "arguments," and "local."**

What are the values of the variables when the program is first at Line 40?

| | |
|---|---|
| `first` | |
| `second` | |
| `previous` | |
| `current` | |
| `count` | |

**→ Then hit the 'Step' button to get the program moving again until it reaches the loop at line 47.**

Now what are the values of the variables when the program is first at Line 47?

| | |
|---|---|
| `first` | |
| `second` | |
| `temp` | |
| `previous` | |
| `current` | |
| `count` | |

**→ Keep hitting the "Step" button and fill in the values below:**

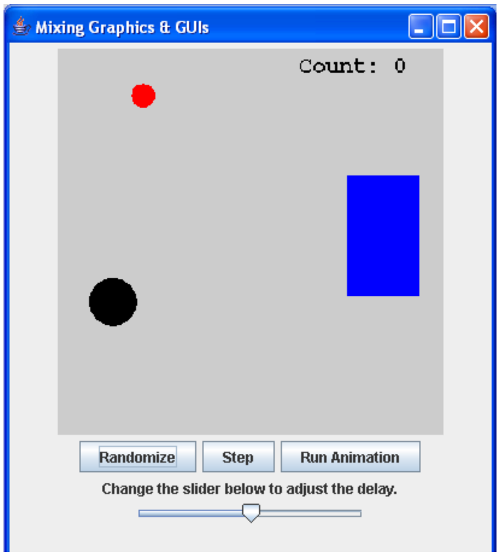
| `temp` | `previous` | `current` | `count` |
|---|---|---|---|
| | | | 2 |
| | | | 3 |
| | | | 4 |
| | | | 5 |

**→ Are the values that you see those that you expected to see? If the program is not calculating as you expect, you can now look for ways to correct the code to get the desired behavior. That is the whole purpose of using the debugger. Using the debugger can be quite helpful.**

***Assignment*** The Lab16 folder has two broken programs, PrimeBroken and HailstoneBroken, with which to practice your debugging skills. You should begin by thinking about the expected output from each program.

# Lab17
## Mixing Graphics and GUIs

### *Objective*
Unit 2 graphics with Unit 3 buttons.



Count: 0

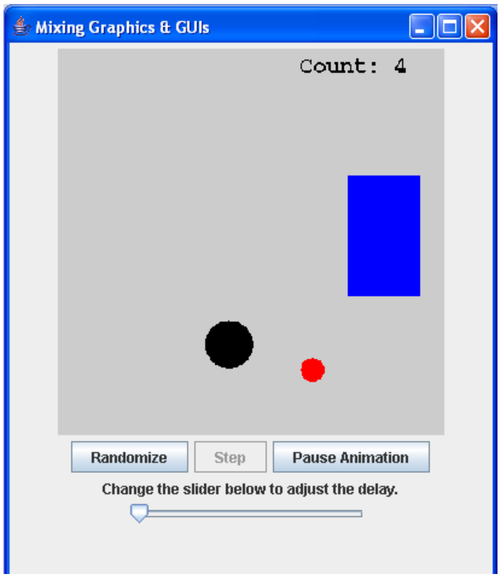
Change the slider below to adjust the delay.

### *Background*
In this lab the JFrame hasa Panel17 object, which hasa BumperPanel object.

Panel17 also has three buttons, a slider, and a timer. The three buttons are registered to three listeners (of course), which call public methods in BumperPanel.

The new BumperPanel is much like the BumperPanel in Unit 2, Lab14. It instantiates a BufferedImage, polkadot, ball, and bumper. It has the methods `paintComponent`, `collide`, and `distance`. It has two new methods `jumpAll` and `stepAnimation`. The latter method does all the work of clearing the panel, moving the balls, checking for collisions, painting all the objects, updating the count, and calling `repaint`.

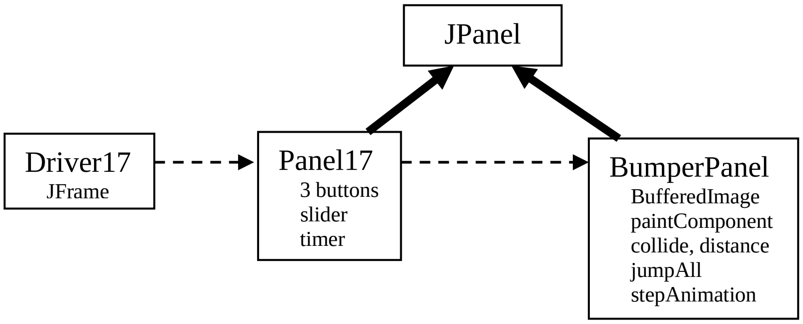
The new BumperPanel does not have a Timer. The timer is instantiated in Panel17. Every time the timer fires, it calls the BumperPanel's `stepAnimation`.



Count: 4

Change the slider below to adjust the delay.

Think about the responsibilities of each class:



When clicked, the "Run Animation" Button changes to "Pause Animation", using an idea from the extension to Unit 3, Lab05.

When the animation is running, the "Step" Button is grayed out. `setEnabled(false)` is a helpful method.

### *Specification*
There are no shells. Write the classes as described above. Good luck.

### *Extension*
Implement the slider to adjust the delay (in milliseconds) of the timer, either slower or faster. JSlider has a helpful command `getValue`. The Timer class has a helpful method `setDelay`.

### *Sample Run*
`Unit3Lab17.jar`