

Objectives	1. Practice creating and using graphical components. 2. Practice adding Event Listeners to handle the events and do something. 3. Learn how to connect a graphical interface to the rest of your application.
What to Submit	

Part I: Learn Common Graphics Components

1. Basic Program Structure

To see the basic layout of the code for a GUI app, create a JFrame containing only a single JButton.

```
import javax.swing.*;  
import java.awt.*;  
import java.awt.event.ActionEvent;
```

Import classes from AWT and Swing

```
public class ComponentDemo {  
    private JFrame frame;  
    // attributes for components  
    private JButton button;
```

Declare attributes for components your application needs to access. Some components can be declared as local vars.

```
    public ComponentDemo( ) {  
        frame = new JFrame("Demo");  
        frame.setDefaultCloseOperation(  
            JFrame.EXIT_ON_CLOSE );  
        initComponents( );  
    }
```

Constructor usually does:

- (1) initialize attributes
- (2) set some properties of the window
- (3) call **initComponents**

```
    /** initialize components in the window */  
    private void initComponents() {  
        // (1) create components for the UI & set properties (color, font ...)  
        // (2) position components using layout managers and containers  
        // (3) add event listeners to components (Problem 2)  
        button = new JButton("Press Me");  
        frame.add( button );  
        frame.pack( );  
    }  
    /** Display the graphics window. */  
    public void run() {  
        frame.setVisible( true );  
    }  
  
    public static void main(String [] args) {  
        ComponentDemo demo = new ComponentDemo();  
        demo.run();  
        System.out.println("Done launching window. Do you see it?");  
    }  
}
```

2. Add an Event Listener

When the user does something in a graphical application, the graphics system dispatches an *Event*. Your program must tell the graphics system what code it should call when an event occurs. This code is called an "event listener" or "event handler" and it must implement a specific interface.

Let's add an *event listener* to the JButton so we are notified when the button is pressed.

2.1 Define an *inner class* that implements *ActionListener* (the interface for button events). The interface has only one method, as shown here.

```
// this class is inside the ComponentDemo class. Its an "inner class".  
class ButtonListener implements ActionListener {  
    /** this method receives events */  
    public void actionPerformed( ActionEvent event ) {
```

```

        System.out.println("Ouch! Don't press so hard");
    }
} // end of inner class

```

2.2 In the `initComponents()` method, add a `ButtonListener` to the button component:

```

/** initialize components in the window */
private void initComponents() {
    button = new JButton("Press Me");
    ButtonListener listener = new ButtonListener();
    button.addActionListener( listener );
    ...
}

```

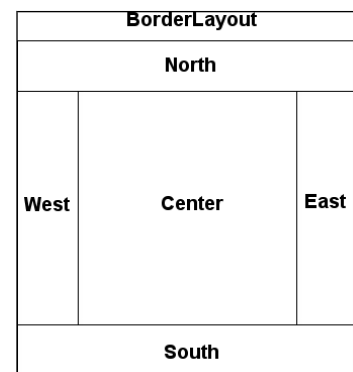
2.3 Test the program -- press the button and see that it works.

3. Layout Multiple Components

Java uses a *Layout manager* to arrange and manage components.

The default layout manager for `JFrame` is `BorderLayout`, which divides window into 5 regions North, Center, East, West, and South.

Components are resized to fill the entire region. To see this, add some labels and a Textfield to the `ComponentDemo` class.



```

public class ComponentDemo {

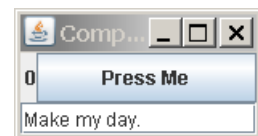
//Add a label and textfield
    private JLabel counter; // count button clicks
    private JTextField message; // display a message

```

```

    private void initComponents() {
        button = new JButton("Press Me");
        ButtonListener listener = new ButtonListener();
        button.addActionListener( listener );
        counter = new JLabel(" 0");
        message = new JTextField("Make my day.");
        frame.add(button); // default location in CENTER
        frame.add( counter, BorderLayout.WEST );
        frame.add( message, BorderLayout.SOUTH );
        ...
    }
}

```



3.1 Count Button Presses

Modify the `ButtonListener` to do:

- count how many times the button is pressed and display the value in counter `JLabel`.
- print a message in the `JTextField` (message) instead of `System.out`.

3.2 Add Your Own Components to the EAST and NORTH.

4. Set Component Properties (*Make your UI Beautiful*)

Most components support these properties:

```

setForeground( Color )
setBackground( Color )
setEnabled( boolean )
setBorder( Border )
setFont( Font )

```

```

foreground color (text color)
background color
true = enable, false = disable
draw a border around component
font for showing text

```



`setToolTipText("press here")` shown when mouse is over component

Text fields like `TextField` also have these properties:

`setEditable(boolean)` false if field is "read only"

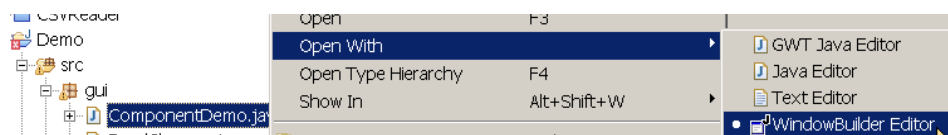
`setHorizontalAlignment(JTextField.CENTER)` LEFT, CENTER, or RIGHT

Visual Graphics Editor for Eclipse

If you are using Eclipse, then use *WindowBuilder Editor* to do this exercise.

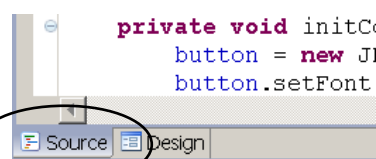
1. Close the editor window for `ComponentDemo`

2. Right-click on `ComponentDemo` and choose "Open with... *WindowBuilder Editor*"

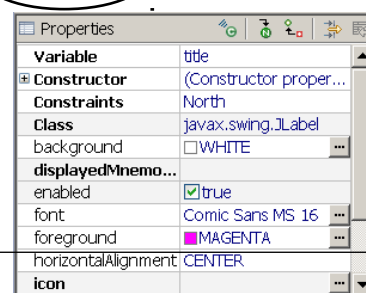


3. At the bottom of the editor window there are 2 tabs: "Source" and "Design".

In Source view you can edit the Java source code. In Design view, use a graphical *Window Builder* to create the UI and set component properties.



4. In Design view, select (click) a component then edit its properties using the Properties panel as shown here.



5. You can switch to Source view at any time and see the code that *Window Builder* has generated for you. You should Save changes before switching views.

How to Create Fonts and Colors in Java: to write the code yourself (instead of using Window Builder), see the end of this handout.

5. Fix Ugly Layout: Arrange Components in a JPanel inside the JFrame

The components look ugly.

`BorderLayout` resizes the components to fill all the space available. We fix this by putting some components in their own container (called a `JPanel`) and then add the `JPanel` to the `JFrame`.

The idea is like this:

1) Create a `JPanel` in `initComponents`.

```
JPanel panel = new JPanel();
```

2) Put components inside the `JPanel` (not frame). The default Layout for `JPanel` is *FlowLayout* so the components will have their preferred sizes.

```
panel.add( label );
panel.add( counter );
panel.add( button );
```

3) Add the `panel` to the `JFrame` (actually it gets added to the `JFrame`'s *contentPane*).

```
frame.add( panel, BorderLayout.CENTER );
```

4) (Optional) Add a Border so that the `JPanel` is *visually distinct*.

```
panel.setBorder( new BevelBorder( BevelBorder.LOWERED ) );
```

Swing has many Border types. Other borders are `EtchedBorder` and `TitledBorder`.



can

6. *Be* a JFrame

In `ComponentDemo` we created a `JFrame` as an *attribute* in the `ComponentDemo` class.

Another way to create a UI is to *extend* the `JFrame` class. Then your UI *is* a `JFrame`.

Both ways ("*has* a `JFrame`" and "*be* a `JFrame`") have advantages, so you should know how to write both. To *be* a `JFrame` requires only a small change in the code you already wrote:

```
public class ComponentDemo extends JFrame {
    private JFrame frame;
    // attributes for components
    private JButton button;

    public ComponentDemo( ) {
        super("Demo"); // set the title
        frame = this; // you don't need the frame variable, but its convenient
        initComponents();
    }
}
```

Since `frame` now refers to **this** object, you can use **this** in your code. For example:

```
/** Display the graphics window. */
public void run() {
    this.setVisible( true );
}
```

7. Try Different Layout Managers

The default layout for `JFrame` is `BorderLayout`. You can change the layout using the command:

```
frame.setLayout( someLayoutManager )
```

Try these layouts and see the difference:

- `GridLayout` arranges components in a grid and makes them the same size

```
LayoutManager layout = new GridLayout( rows, columns ); //try: (2,3)
frame.setLayout(layout);
frame.add(button);
frame.add( new JLabel("Express Yourself") );
frame.add( new JTextField(10) );
...
```



- `FlowLayout` *flows* components to use available space and does not resize components. This is the default layout for `JPanel`.

```
LayoutManager layout = new FlowLayout( );
// other code same as above
```

- `BoxLayout`: Horizontal layout is `BoxLayout.X_AXIS`, vertical layout is `BoxLayout.Y_AXIS`.

```
layout = new BoxLayout( contents, BoxLayout.X_AXIS );
```

- `GridBagLayout`: This is one of the best, most flexible layouts and not *too* hard to use, but you need to study it a little. On your own time, please read the Java Tutorial for `GridBagLayout`.

Part II: Add a GUI as a User Interface in the GuessingGame

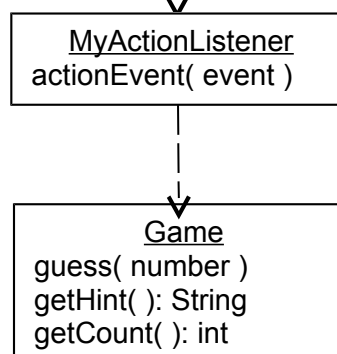
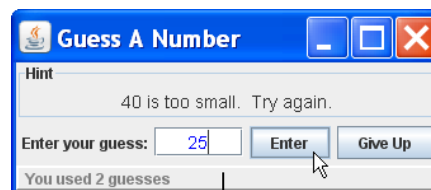
In the Guessing Game you divided the game into 2 classes: Game (the game itself) and GameConsole (user interface). You *assigned different responsibilities to separate classes*. A benefit of this is that you can change the user interface and still reuse the same Game code.

Apply this design to create a Graphical UI for the Guessing Game.

User Interface or View: the top layer handles interaction with the user, manages the display and receives input *events*.

Controller handles processing of user requests. It knows how the application should behave; e.g. stop when the user guesses the secret. It also knows how to translate *user input* into method calls for the domain layer.

Domain Layer or Model contains logic used by application, provides "domain objects" and services.

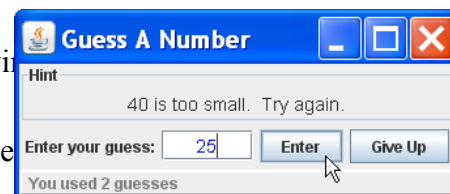


1. Create a GUI Interface for the GuessingGame

Use what you learned in Part 1 of this lab to create a GUI for playing the guessing game.

The GUI needs a *reference* to the game, just like the GameConsole interface you wrote before:

```
GameUI gui = new GameUI( game );
```



WRONG: The GameUI should not create a Game.

```

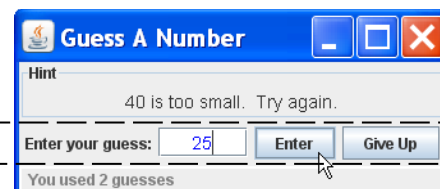
class GameUI extends JFrame {
    private Game game;
    public GameUI( ) {
        // WRONG. Don't write this. UI does not create domain objects.
        game = new Game(100);
    }
}
  
```

2. Use JPanel to Separate the Input Components

Example:

```

JPanel panel = new JPanel();
panel.add( label ); // a label
panel.add( inputField );
panel.add( enterButton );
panel.add( quitButton );
contents.add ( panel, BorderLayout.CENTER );
  
```



3. Write a Main Class to Connect Components Together

Just like in the original Guessing Game, write a separate Main class to create the objects and connect them together.

Common Graphics Components

JLabel displays text and/or an image:

```
JLabel label = new JLabel("A nice label");  
ImageIcon icon = new ImageIcon("c:/images/1baht.png");  
JLabel label = new JLabel(icon);
```

JButton displays text and/or an image. You can press it.

```
JButton button = new JButton("Press Me");  
ImageIcon coin = new ImageIcon("c:/images/1baht.png");  
JButton button2 = new JButton("One Baht", coin);
```



JTextField for inputting text:

```
JTextField textfield = new JTextField( 12 ); // 12 is width
```

JComboBox to select one item from choices

```
// you can put any kind of Objects in a combo box  
String [] fruit = {"Apple", "Banana", "Orange"};  
JComboBox comboBox = new JComboBox( fruit );
```

JTextArea is a text box with more than one row:

```
JTextArea textarea = new JTextArea( rows, columns );  
textarea.setLineWrap(true);  
textarea.setWrapStyleWord(true);
```

JSlider has a min and max. It can also have labels and tick marks. (See Java Tutorial)

```
JSlider slider = new JSlider( 0, 100 ); // min 0, max 100  
slider.setValue( 50 ); // current value  
slider.setMajorTickSpacing( 25 );  
slider.setPaintLabels( true );
```

The Java Tutorial has more component examples:

<http://docs.oracle.com/javase/6/tutorial/ui/features/components.html>

How to Create a Color

The Color class has predefined colors: Color.RED, Color.BLUE, Color.GREY, Color.CYAN, etc.

You can create your own colors using:

```
Color color = new Color( red, green, blue) // red, green, blue are int between 0 and 255  
Color darkgreen = new Color( 0, 100, 0); // red = 0, green = 100, blue = 0  
Color orange = new Color( 255, 165, 0); // maximum red + some green  
Color torquiose = new Color( 0, 245, 255); // green and blue
```

How to Create a Font

```
Font font = new Font( name, style, size );  
Font arial = new Font("Arial", Font.PLAIN, 24);
```

name String font name like "Arial", "Angsana New", or an *OS-independent* font name: Font.DIALOG, Font.DIALOG_INPUT, Font.MONOSPACED, Font.SERIF, or Font.SANS_SERIF

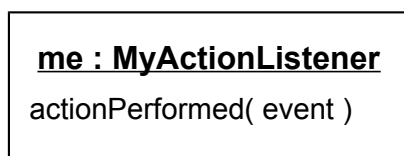
style one of Font.PLAIN, Font.BOLD, Font.ITALIC. (integer constants)

size font size in points. Like in a word processor.

Event Listeners

Use an *Event Listener* so that your code will be notified when something happens in the graphical interface.

1) create an ActionListener



2) add Listener to a Button

```
button.addActionListener( me );
```



3) wait for an event

Click!



4) Java calls your code

```
actionPerformed( ActionEvent evt )
```

Writing an ActionListener

There are 3 common ways to write event listeners:

- create an *inner class* in the GUI that *implements* the event listener. The advantage of an *inner class* is that it can access members its outer class, *even private attributes and private methods*.
- create an *anonymous class* and immediately assign it to a variable or component as a listener
- use the GUI class itself as the event listener; e.g. declare the GUI *implements ActionListener*

Here is how to write an *inner class* to implement *ActionListener*. This example displays info about the event on the console so you can see what is passed as the *ActionEvent* parameter.

```

class ButtonListener implements ActionListener {
    private int count = 0; // count the events
    public void actionPerformed((ActionEvent event) {
        // which component caused the event?
        Object source = event.getSource(); // e.g. a JButton object
        System.out.println(
            "Event source: " + source.getClass().getName());
        // what event or command occurred?
        String cmd = event.getActionCommand();
        System.out.println("Action command: " + cmd);
        // count the event and show the count
        count = count + 1;
        label.setText( Integer.toString(count) );
    }
}
  
```

You must add the *ActionListener* to your components. You can use the *same* listener object as listener for several components.

```

ActionListener listener = new ButtonListener();
button.addActionListener( listener );
textfield.addActionListener( listener ); // reuse same listener
  
```

Other Kinds of Event Listener

Different components require different kinds of event listeners. For example, a *JSlider* uses a *ChangeListener*. To write a *ChangeListener* for a *JSlider*:

```
/** a listener for slider */
```



```
class SliderListener implements ChangeListener {  
    public void stateChanged(ChangeEvent e) {  
        JSlider slider = (JSlider) e.getSource();  
        int value = slider.getValue();  
        System.out.println( "slider value is "+ value );  
    }  
}
```

In your `initComponents()` method you would add a change listener to a slider using:

```
slider.addChangeListener( new SliderListener() );
```

The Recommended Way to Launch Swing Applications

Swing uses threads. This is why a GUI window keep running even after your `main()` method has finished. Swing may use several threads, but one thread is special: the *Event Dispatcher Thread*. This thread performs event handling, which includes responding to all user input.

Java has **two rules** for using threads in Swing:

1. all UI updates should be performed **only** in the *event dispatcher thread*.
2. time consuming or long running tasks should **not** be performed in the *event dispatcher thread*.

Rule 1 means we should start our Swing UI in the event dispatcher thread. But how?

The `SwingUtilities` class has method that will insure your code is run in the event dispatcher thread. The typical way of using it is this:

```
public static void main(String [] args) {  
    /* Create and display the form */  
    SwingUtilities.invokeLater( new Runnable() {  
        public void run() {  
            new ComponentDemo().setVisible(true);  
        }  
    });  
}
```

`SwingUtilities.invokeLater()` starts the *Runnable* and returns immediately. As an alternative, you can call `SwingUtilities.invokeAndWait()` which will wait until the *Runnable*'s `run()` method exits.

In cases where you need to apply Rule #2 (run a task on another thread and communicate the results to the event dispatcher thread so they show on the UI), use the `SwingWorker` class.