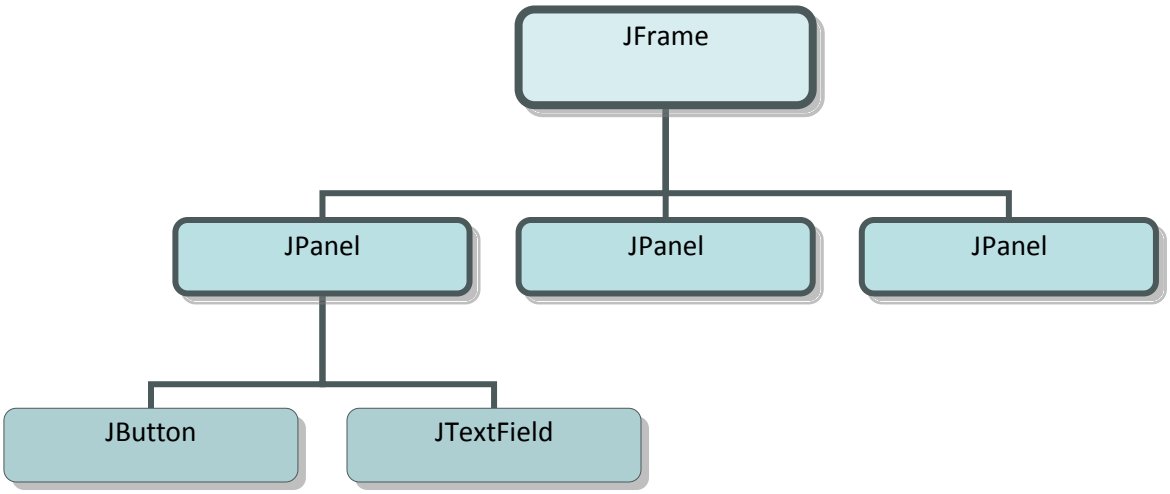
Introduction to Graphical User Interfaces (GUIs)

A graphical user interface, typically shortened to GUI (pronounced G-U-I or G-OO-E), is a simple form of user interface that allows people to interact with a computer. Usually a GUI in Java is composed of the window itself, a `JFrame`, which contains various panels, `JPanel`s. The `JPanel`s in turn can contain any number of components such as buttons and text fields.



The relationship between the parts of a GUI where a JFrame contains JPanels. A JPanel contains components such as JButtons and JTextFields.

Arranging the Components: Layout Managers

You can control the layout of a `JPanel` by using the `LayoutManager`s. There are numerous layouts but the following are the primary ones we would like you to focus on:

`FlowLayout`, typically just called *flow* when discussing `LayoutManager`s, is the default layout that simply adds content from left to right and centering the row within the panel.

`BorderLayout`, or just *border* for short, allows you to specify what region of the panel NORTH, SOUTH, EAST, WEST, and CENTER to add the component. Remember each region can only contain one component.

`GridLayout`, or simply *grid*, allows you to add components in a grid or table-like fashion, specifying the number of rows and columns when creating it.

You can specify the layout of the panel in one of two ways:

1. Using the `setLayout` method.

```
JPanel myFlowPanel = new JPanel();
JPanel myBorderPanel = new JPanel();
JPanel myGridPanel = new JPanel();
```

```
//Unnecessary step because default layout is flow
myFlowPanel.setLayout(new FlowLayout());
myBorderPanel.setLayout(new BorderLayout());
//3 is the number of rows, 4 the columns
myGridPanel.setLayout(new GridLayout(3, 4));
```

2. Passing in a layout into the constructor of `JPanel`

```
JPanel myFlowPanel = new JPanel(new FlowLayout());
JPanel myBorderPanel = new JPanel(new BorderLayout());
JPanel myGridPanel = new JPanel(new GridLayout(3, 4));
```

Look here for more information on layout managers:
http://java.sun.com/docs/books/tutorial/uiswing/layout/using.html

All the little parts: GUI Components

Now that you have some understanding about how GUIs are laid out, you are probably wondering how we put labels, buttons, and text fields into a `JPanel`. When you declare a *`JLabel`*, all you need to give it is the text you want to show on it.

```
JLabel theLabel = new JLabel("Some Text");
```

Once you have a `Jlabel` you can add it to the `JPanel` like this:

```
myPanel.add(theLabel);
```

Or if you are using a `BorderLayout`, you must specify the position:

```
myPanel.add(theLabel, BorderLayout.NORTH);
```

*`JButton`*s are again very similar in declaration, because they take in text that will be on the button.

```
JButton theButton = new JButton("Some Text");
```

*`JTextField`*s are small text boxes where the user can enter input. When creating a `JTextField`, it takes not only an initial string to display but also the desired width of the field.

```
JTextField aTextField = new JTextField("Initial Text", 20);
```

Making Components Function with the User: Listeners

If you have tried adding buttons and text fields to your GUI, you may have noticed that it does not really do a whole lot at this point. This is because you must add an `ActionListener` to it. An *`ActionListener`* is what sits around waiting for something to happen, such as a button being pressed or a text field being used. You must create your own classes that implements

`ActionListener`. Although you can create a separate listener class for each button and each text field, this seems to be quite inefficient and messy when you have lots of buttons or fields. Luckily there is a way to distinguish the source of the action listener calls and thus you can specify what to do for each component within the same action listener.

`MyButtonListener`

The first listener class we will call `MyButtonListener` which implements `ActionListener` and overwrites the method called `actionPerformed` from the `ActionListener` class. As mentioned above, you can use the `getSource()` method from the `ActionEvent` to learn what the component with an `ActionListener` is being used of the user.

```
private class MyButtonListener implements ActionListener {
     public void actionPerformed (ActionEvent event){
          Object source = event.getSource();
```

Now, within this method you can use a series of if else statements, testing to see which button called it.

```
          if(source.equals(buttonA))
               //do something
          else if(source.equals(button))
               //do something else
```

A very similar approach can be used for the class `MyTextListener`.

Now that we have created our listeners, you can add them to the corresponding buttons and text fields by the `addActionListener()` method like so:

```
      buttonA.addActionListener(new MyButtonListener());
```

`MyKeyListener`

`MyKeyListener` will take input from the keyboard and perform specified functions. The constructor is given to you:

```
  private class myKeyListener extends KeyAdapter {
    public void keyPressed (KeyEvent event){
```

`KeyEvent` has a method similar to `getSource()` from earlier called `getKeyCode()` which returns an integer corresponding to which key was pressed. Some integer values you may find helpful are:

Integer value	Corresponds with
KeyEvent.VK_UP	Up arrow key
KeyEvent.VK_DOWN	Down arrow key

KeyEvent.VK_LEFT	Left arrow key
KeyEvent.VK_RIGHT	Right arrow key

Creating GUIs with Style: Model-View-Controller (MVC)

Model-View-Controller is a style of creating GUIs that can make a developer's job much simpler in the long run. The general idea is that you should split up your GUI into three parts: the model, the view and the controller.

The Model

The *model* is what we call the "back end" of the program. It is only responsible for the behind-the-scenes jobs. In our Etch-A-Sketch, it will maintain an instance of our World and the Turtles on it. This is where we will directly manipulate properties such as where the turtles are on the screen, pen width, pen color, speed of drawing, or exporting/importing to/from a file. In general, the model should be a stand-alone program that can be "hooked up" to any interface without being changed.

The View

The *view* is the part that displays information to the user. The panels in the GUI are an example of this.

The Controller

The *controller* refers to the part that takes in information from the user and often times coupled with the View. Action Listeners are in this category because they take mouse clicks and text from the user and pass it along to the Model.

The primary advantage of MVC is reusability of code. You should be able to take the Model and put any number of "front ends" on it to display the information in different ways.