The approved way to do a GUI application is to use the View-Model-Controller Paradigm.

We will start off much more simply. You may need these imports:

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

Here is a simple Hello world main program (note that you must supply the usual Java class structure yourself - a good class name is Application):

```java
public static void main(String[] args) {
    JOptionPane.showMessageDialog(null, "Hello World");
}
```

A class for super easy popup GUI stuff

Convenient Factory Method which also activates resulting JOptionPane object

There is no Frame being used.

Message to be displayed in the popup dialog

The usual main method stuff
A JOptionPane object can have many other things in it besides just a text message.

For example - you can put a button in place of the message.

First of all, you must create the button (do this in your main method):

```java
JButton mybtnjoe = new JButton("Press Easy");
```

- **Class for Buttons**
- **Local variable to hang on to newly created JButton object**
- **Caption that will appear on top of button once your application starts.**
Once you have a button object, use it inside the `showMessage` method in place of the message string.

```java
public static void main(String[] args) {
    JButton mybtnjoe = new JButton("Press Easy");
    JOptionPane.showMessageDialog(null, mybtnjoe);
}
```

**NOTE:** Pressing your button WILL NOT do ANYTHING YET (except change shape slightly).

Later you will require learn about `ActionListener` objects.
Buttons are examples of object that we will call **widgets**.

Two other kinds of simple widgets are *labels* and *text entry boxes*.

The `javax.swing` class names for these are:

- `JButton`
- `JLabel`
- `JTextField`

Generically, the constructors for these widgets are:

- `new JButton( string_giving_caption_on_button )`
- `new JLabel( string_giving_message_widget_displays )`
- `new JTextField( number_roughly_giving_width_in_characters )`
A JOptionPane object can display more than one of these widgets.

All that is needed is that the desired widgets are placed in an array of Objects:

Below is an example showing a label, button, and entry box. The example just gives the commands - you need to supply your own class and method.

```java
JLabel wgta = new JLabel("Hi There");
JButton wgtb = new JButton("Hit ME");
JTextField wgtc = new JTextField(10);
Object[] mywidgets = { wgta, wgtb, wgtc };  
JOptionPane.showMessageDialog(null,mywidgets);
```

You need to write your own Java application to try out the above commands.
An ActionListener is an object with a method named `actionPerformed`.

HOWEVER, it is an Interface (namely very abstract) and actually has no such method provided.

Thus, the programmer must SUBCLASS this class and PROVIDE the *actual* `actionPerformed` method.

Here is an example of such a class file (use the imports mentioned earlier)

class AddsThree implements ActionListener {
    int myvalue = 10; // an instance can have state

    public void actionPerformed(ActionEvent e) {
        System.out.printf("value currently is \d\n",myvalue);
        myvalue += 3;
    }
}
ActionListeners really are objects. They are created with a `new` command and can have their methods called.

Below are some steps that could go in a main method and would exercise the `AddsThree` class shown in the previous slide.

```java
AddsThree thingx = new AddsThree();
AddsThree thingy = new AddsThree();
thingx.actionPerformed(null);
thingx.actionPerformed(null);
thingy.actionPerformed(null);
```

The output from these steps should be:

```
value currently is 10
value currently is 13
value currently is 10
```

However, this does NOT MAKE a GUI program.
As you saw in the "NonFrame Buttons-2" slide, clicking the button displayed in the JOptionPane did not have any effect.

Buttons acquire action by having an ActionListener object ADDED to them.

So assuming the AddsThree class is in your project, you could put these steps into your main method:

```java
JButton mybtn = new JButton("Hit to Add");
ActionListener act = new AddsThree();
mybtn.addActionListener(act);
JOptionPane.showMessageDialog(null, mybtn);
```
Typically a different ActionListener subclass is used for each different button in a GUI application.

Making up names for each of these subclasses becomes irritating.

Fortunately a new operation and a subclass declaration can be done in a single command.

```java
int myvalue = 10;
System.out.printf("value currently is %d\n",myvalue);
myvalue += 3;

public void actionPerformed(ActionEvent e) {};

ActionListener act = new ActionListener() {
    int myvalue = 10;
    public void actionPerformed(ActionEvent e) {
        System.out.printf("value currently is %d\n",myvalue);
        myvalue += 3;
    }
};
```

The next slide will break this up and explain the parts.
Variable to hold new action object

Create a new instance

Class to be subclassed

Missing subclass name

ActionListener act = new ActionListener() {
    int myvalue = 10;

    public void actionPerformed(ActionEvent e) {
        System.out.printf("value currently is %d
\n", myvalue);
        myvalue += 3;
    }
}

End of assignment statement which stored the newly created object into the variable act.

Declarations for subclass

Extension of subclass
Actually, the ActionListener object variable `act` will often only be used once, and so both the ActionListener creation command and the attachment of the action to the button can be combined into one highly confusing command:

```java
mybtn.addActionListener(
    new ActionListener() {
        int myvalue = 10;
        public void actionPerformed(ActionEvent ev) {
            System.out.printf("value currently is %d\n", myvalue);
            myvalue += 3;
        }
    }
);
```

- Attach action to button
- Declarations of subclass
- Creation of ActionListener object
- End of call to addActionListener method
- Add Anonymous
The actions attached to different buttons usually want to share in the state of an object.

Buttons created by a static method can only share a static class variable.

Buttons created by a class instance method can share not only the static class variables but also the instance variables.

The next slide has an example of such a program.
class HasTwoMembers {
    static int changes = 0;
    int width = 3;
    int height = 10;
    void activate() {
        JButton moreWidth = new JButton("++ Width");
        moreWidth.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    width += 8;
                    ++ changes;
                } }); // lazy combination of three code lines
        JButton lessHigh = new JButton("Decr Height");
        lessHigh.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    height /= 2;
                    ++ changes;
                } });
        JOptionPane.showMessageDialog(null,
            new Object[] { moreWidth, lessHigh } );
    }
    void dump() {
        System.out.printf("object width=%d height=%d\n",width,height);
    }
    static void printChanges() {
        System.out.printf("Total instance changes=%d\n",changes);
    }
    public static void main(String[] args) {
        HasTwoMembers joe = new HasTwoMembers();
        HasTwoMembers mary = new HasTwoMembers();
        joe.dump();
        joe.activate();
        joe.dump();
        printChanges();
        mary.dump();
        mary.activate();
        mary.dump();
        printChanges();
    }
}
So, a JButton widget is used by attaching a customized ActionListener object to it.

How, then, is a data entry widget [namely a JTextField object] used.

This is easier than a JButton.

A JTextField is kind of like a VISIBLE String variable.

So suppose we have created such an object - for example:

```java
JTextField myvisvar = new JTextField(5);
```

To "store" a string value into this entry use the setText method in a command like:

```java
myvisvar.setText(the_string_value);
```

To "access" a string value from this entry use the getText method in a command like:

```java
String z = myvisvar.getText();
```

**NOTE: NOTE: NOTE:**
The two methods setText and getText are how the program accesses the data in the entry. The user simply looks at the entry widget to see its value. The user simply clicks in the entry and enters text information in it to "store" values into the widget.
Here is some main method code to popup a dialog to get some text from the user. The code then decides whether it likes the text or not - usually it does not.

```java
    JTextField anEntry = new JTextField(20);
    Object[] content = { "What do you like?", anEntry);
    JOptionPane.showMessageDialog(null, content);
    String userSaid = anEntry.getText();
    int xfound = "chocolate coffee juice".indexOf(userSaid);
    if(xfound>=0)
        System.out.printf("I like %s too.",userSaid);
    else
        System.out.printf("I hate %s!",userSaid);
```

Create Entry widget

content for popup

Display popup, wait for its OK

Transfer data from entry to variable userSaid

trick for multiple comparison

Program's response
A JTextField entry widget acts like a String variable.

If the user types a number into the entry, then it is kept as a String.

So if the user types the number 365 into the entry, then the value in the entry is the String "365". It is NOT the integer 365.

Even worse, if the program tries to put an integer into the entry using the setText method, the program will not compile.

Here are three commands which illustrate how to get around these problems - where it is assumed that theData is a JTextField widget to be used for integer data entry and that myans is an int variable.

**Get integer from entry widget into variable value.**

```java
int value = Integer.parseInt(theData.getText());
```

**Display integer myans in entry widget - slow version**

```java
theData.setText( "" + myans);
```

**Display integer myans in entry widget - faster version**

```java
theData.setText(Integer.toString(myans));
```