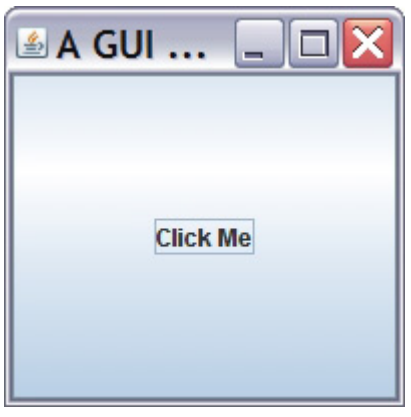
# Chapter 3: Buttons and Interaction

## Buttons and Interaction

Next, you're going to learn how to display an item that allows us to interact with it. We'll start by displaying a button, then we'll see how we can make it do something. Displaying a button is quite a bit like displaying our text. Here's the program, followed by the window it displays:

```
import java.awt.*;

import javax.swing.*;



public class GUIButton1

{

  private JFrame frame;



public static void main (String[] args)

{

  GUIButton1 guiButton = new GUIButton1();

  guiButton.start();

}



public void start()

{

  frame = new JFrame("A GUI Button");

  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  Container contentPane = frame.getContentPane();



  JButton button = new JButton("Click Me");

  contentPane.add(button);



      frame.setSize(200,200);

      frame.setVisible(true);

    }

}
```

Here's the window:




A window with a button

As you can see, the text for the button can be set in its constructor, just like the label in the previous example. But there's something odd about this button: It takes up the whole window! It's a very big button. And it might not seem to do anything. If you run this program, you'll notice that it does do *something*—when you move your mouse over it, Java highlights the outline of the button. And if you click it, the color changes. But it doesn't do anything else.

So how do we make our button the size that we want? And how do we get it to do what we want? Those are two separate issues that we'll need to deal with individually.

First, let's talk about the size. The size of a button automatically expands to fill up the pane. Since we set the pane's size at 200 x 200, that's the size the button became. We can reverse the action and make the window the size of the button by replacing the call to setSize(200, 200) with a call to the frame's pack() method, like this:

```
frame.pack();
```

The pack() method changes the size of the window to fit whatever components are in it. When we replace the setSize (200,200) call with this, here's how our window looks:


A packed frame

If you're wondering how to get a small button in a large window, don't worry—we'll get to that later. We need a little more background first. So let's find out how to make our button do something. We'll make it simple. Let's say that if someone clicks our button, we want the color of the text to change to red and the message on the button to change to "I've been clicked!"

The code is the easy part. A little digging in the Java API will turn up the method calls we need to do that:

```
button.setForeground(Color.red);

button.setText("I've been clicked!");
```

But where do we put these calls? How can our program know when the button gets clicked?

Luckily, Java has a mechanism for dealing with GUI user actions. It's called *event-handling*. A Java *event* is a user action in the GUI that needs to communicate with the program. There are lots of different events in Java, and one of them is a button click. How do we get the button to tell us if it's been clicked? We have to *listen* for it.

I don't mean listen for an audible sound, though. Java has a way for us to tell a GUI component that we want to listen for events that happen to it. We do that by implementing an interface. (Remember those from the last lesson?)

Java has several types of listener interfaces. There's the KeyListener, the MouseListener, the TextListener, and more. The java.awt.event package lists all of them. The one we want is the *ActionListener* interface.

ActionListener has one method: actionPerformed(). If we use the ActionListener interface to tell the button that we want to listen for its actions, then it will let us know when an action happens by calling our actionPerformed() method. The actionPerformed() method is where we need to put the code we want to execute when the button is clicked. We do it like this:

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


public class GUIButton3 implements ActionListener
{
  private JFrame frame;
  private JButton button;


  public static void main (String[] args)
  {
    GUIButton3 guiButton = new GUIButton3();
    guiButton.start();
  }


public void start()
{
  frame = new JFrame("A GUI Button");
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  Container contentPane = frame.getContentPane();


  button = new JButton("Click Me");
  button.addActionListener(this);
  contentPane.add(button);


     frame.pack();
     frame.setVisible(true);
  }


    public void actionPerformed(ActionEvent e)
    {
       button.setForeground(Color.red);
       button.setText("I've been clicked!");
    }
}
```

If you run this program, it will look like the last figure when it starts. As soon as you click the button, it will change to this:


A clicked button

To make this work, besides adding the actionPerformed() method and the calls it contains, we made four other changes to the program from last time. First, since the ActionListener interface is in the java.awt.event package, we needed to import that (see line 2).

Second, we added *implements ActionListener* to the class heading. That tells Java that we'll be implementing the actionPerformed() method for the button to call.

Third, since we needed to use the variable button in two methods, we moved its declaration out of the start() method and made it an instance variable. We now declare it in the second line of the class body, right after frame.

Last, we told Java to add us to the button's listeners. That's the call to addActionListener() in the fifth line of the start() method. The argument tells Java that our program will listen for actions connected to the button.

Phew! We've learned a lot already, but I have one more thing to share before we move on. Whenever an event starts a listener, it always passes an event argument to give us information about the event and its source, in case we want to use it. In this example we don't need it, but it's still there in our program because that's how the ActionListener interface defines the actionPerformed method. So we created an ActionEvent parameter, and we named it *e*. Later, we'll see events where we'll want to use information from the argument, but we can ignore it this time.