VidWorks Entertainment New Developer Training Program
Tutorial #3

# User Interface

NOTE: Java 1.5.0 is highly recommended for this and all subsequent tutorials.

**Swing, AWT, and the JApplet**

Up until now, we have been implicitly using the Java API called the Abstract Windowing Toolkit (hence the `java.awt` package we've been importing from). However, now is the time to switch to a much more powerful and much preferred Swing API. However, in order to do so, we must first introduce the JApplet, Swing's own version of the Applet, using some example code to illustrate the differences:

```
/* JHelloApplet.java */

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

public class JHelloApplet extends JApplet {
      public void init() {
            JHelloPanel myPanel = new JHelloPanel();
            setContentPane(myPanel);
      }
}

class JHelloPanel extends JPanel {
      protected void paintComponent(Graphics g) {
            if (isOpaque()) {
                  g.setColor(getBackground());
                  g.fillRect(0, 0, getWidth(), getHeight());
            }

            g.setColor(getForeground());
            g.drawString("Hello, World!", 30, 30);
      }
}
```

As we see here, the Applet and the JApplet have different painting models. In an Applet, everything is painted directly to the Applet. However, it turns out that the JApplet has multiple layers, so we can't paint directly to the JApplet. Instead, we paint to what's called the JApplet's content pane. What we do instead is, we extend a JPanel and paint to that, then set it as the JApplet's content pane.

Note that, unless we do a custom paint job to the content pane, we don't have to create our own subclass to JPanel and replace the JApplet's content pane. Today, we'll see some examples where the default content pane is perfectly adequate.

The other big difference is that rather than painting in the `paint()` method, we use a method called `paintComponent()`. In addition, Swing components don't automatically paint the background for us, so we must check if the component is opaque, and if so, paint its background. If we don't, since some JVMs optimize painting, if we

don't paint our background, there will be no guarantee what is painted underneath our JPanel, and we could potentially end up with garbage.

In any case, here is our output:


Notice that it looks slightly different from our original Hello World applet (most notably the background color in our screenshots). This is due to inherent differences in the Swing and AWT architectures.

So if Swing is supposedly so much better than AWT, then why did I start with teaching you the AWT Applet? Because AWT is simpler, and therefore, makes it a better teaching tool. As we saw, Hello World was a one-liner in AWT, but it became so much more complicated in Swing. In addition, the first two tutorials were to focus on painting and event handling, and I didn't want to clutter things up with Swing until after we were ready to do components.

Or you could say that it slipped my mind. It doesn't really matter.

**Swing Components**

Note that this tutorial is *not* meant to be a comprehensive guide to Swing. Swing is far too vast for a tutorial to cover--or even 2 tutorials. Instead, we're just going to take a brief tour through various Swing concepts to allow you to get the feel of what Swing can do. However, it will only be through your own initiative that you can discover all of what Swing has to offer.

**Swing Components - Labels**

Swing provides components that you can use to display information without having to do any painting. One such component is the `JLabel`. Here are some useful methods:

`JLabel(String text)` - constructor that creates the label with the given text
`String getText()` - gets the label's text
`void setText(String text)` - sets the label's text

And below is the Hello World example using labels:

```
/* HelloLabelEx.java */

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

public class HelloLabelEx extends JApplet {
      public void init() {
            setLayout(new FlowLayout());
            add(new JLabel("Hello, World!"));
      }
}
```

Much simpler, isn't it? Oh, and our output, given a 300x200 applet, is this:


One note: the `setLayout(new FlowLayout())` line. This sets the layout of the JApplet to the FlowLayout, which is one of the simpler layouts of Java. Layouts will be covered in a later tutorial. For now, just understand that the Flow Layout simply adds things to the applet in the order the calls to the add method appear.

Note that we call the `add()` method to add components to our JApplet. However, in version 1.4.2 and earlier, you have to use the more cumbersome "`getContentPane().add(/*stuff*/)`". If you have an older version of Java, either upgrade it or be sure to use `getContentPane().add()` instead.

**Swing Components - Buttons**

Swing provides a class called `JButton`, which can be used as a button on your program. Here are some useful methods:

`JButton(String text)` - the constructor, it creates a button with the given text as its label
`String getText()` - gets the text of the button
`void setText(String text)` - sets the text of the button
`int getMnemonic()` - gets the mnemonic of the button. A mnemonic is a keyboard shortcut that triggers a button push. If the mnemonic is M, for example, then Alt+M triggers the button. The mnemonic is in the form of a KeyEvent constant, for example, VK_M. Note that mnemonics are case-insensitive
`void setMnemonic(int mnemonic)` - sets the mnemonic
`String getToolTipText()` - gets the tool tip text of the button, if any. A tool tip is the helper text that appears when the mouse hovers over the button.
`void setToolTipText(String text)` - sets the tool tip text. Pass null to the method to turn off the tool tip
`boolean isEnabled()` - returns whether or not the button is enabled
`void setEnabled(boolean b)` - enables or disables the button. A button is, by default, enabled.

In addition, you may set *one* of the buttons on your applet to be the default button, which means it gets highlighted and it'll usually trigger when the user hits Enter. To set the default button, you'll first need to get the root pane of the applet by calling `getRootPane()` on the JApplet. The root pane has the following methods that allow you to get and set the default button:

```
JButton getDefaultButton()
void setDefaultButton(JButton button)
```

Now, we want to know whenever the user clicks the button. For that, we'll use the `ActionListener` interface, which has a single method:

```
void actionPerformed(ActionEvent e)
```

Sometimes, however, there will be multiple buttons on an applet, which is why ActionEvent has the following method:

```
Object getSource()
```

Comparing the object returned by `getSource()` to your buttons will help you determine which button was clicked. Notice that `getSource()` returns a generic Object, which suggests (correctly) that not only buttons generate Action events.

Here's a simple example that demonstrates buttons:

```
/* JButtonDemo.java */

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

public class JButtonDemo extends JApplet implements ActionListener {
      JButton bx, by;
      JLabel lx, ly;
      int countx, county;

      public void init() {
            setLayout(new FlowLayout());

            bx = new JButton("Button X");
            bx.setMnemonic(KeyEvent.VK_X);
            bx.setToolTipText("This is a button");
            bx.addActionListener(this);
            add(bx);

            by = new JButton("Button Y");
            by.setMnemonic(KeyEvent.VK_Y);
            by.setToolTipText("This is a button");
            by.addActionListener(this);
            add(by);

            lx = new JLabel("Button X has been pushed 0 times");
            add(lx);

            ly = new JLabel("Button Y has been pushed 0 times");
            add(ly);
      }

      public void actionPerformed(ActionEvent e) {
            Object source = e.getSource();
            if (source == bx) {
                  countx++;
                  lx.setText("Button x has been pushed " + countx + "
times");
            } else if (source == by) {
                  county++;
                  ly.setText("Button x has been pushed " + county + "
times");
            }
      }
}
```

Using a 250x100 applet, these are some screenshots of our output:


**Swing Components - Text Fields**

Buttons aren't the only way for the user to interact with the applet. Swing also provides text field components for the user to enter text, the simplest of which is the `JTextField`. Some of its useful methods include:

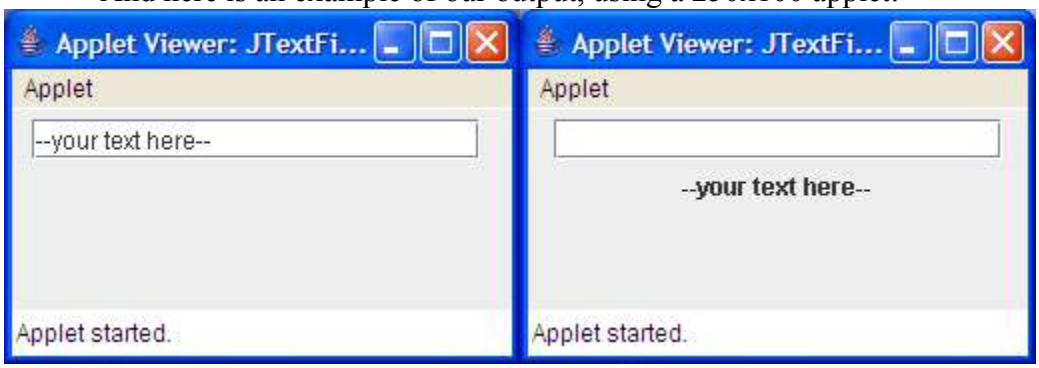
`JTextField()` - the default constructor with no initial text and number of columns set to zero
`JTextField(String text)` - a constructor that provides initial text, but no number of columns
`JTextField(int columns)` - a constructor that provides number of columns, but no initial text
`JTextField(String text, int columns)` - a constructor that provides both initial text and number of columns
`String getText()` - gets the text field's text
`void setText(String text)` - sets the text field's text
`int getColumns()` - gets the number of columns the text field has
`void setColumns(int columns)` - sets the number of columns the text field has. Note that the number of columns only affects the text field's preferred size. A column size of zero means that the text field has no preferred size. Preferred sizes are not always honored, depending on layout. However, FlowLayout will *usually* honor preferred sizes.
`boolean isEditable()` - returns whether or not the text field is editable
`void setEditable(boolean b)` - makes a text field editable or non-editable. Text fields are, by default, editable.

Text fields, like buttons, can fire Action events. When the user presses Enter while in the text field, this will trigger an action event. We use the same Action listener as before.

And now, for a simple example:

```
/* JTextFieldDemo.java */

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

public class JTextFieldDemo extends JApplet implements ActionListener {
      JTextField tfield;
      JLabel echo;

      public void init() {
            setLayout(new FlowLayout());

            tfield = new JTextField("--your text here--", 80);
            tfield.addActionListener(this);
            add(tfield);

            echo = new JLabel("");
            add(echo);
      }

      public void actionPerformed(ActionEvent e) {
            if (e.getSource() == tfield) {
                  echo.setText(tfield.getText());
                  tfield.setText("");
            }
      }
}
```

And here is an example of our output, using a 250x100 applet:



**Swing Components - Check Boxes**

Another useful Swing component is the JCheckBox. Both JCheckBox and JButton were derived from the superclass AbstractButton, so both have many of the same methods, including the methods we've already covered for getting and setting text, mnemonics, tool tips, and for disabling/enabling.

JCheckBox has the following constructors:

`JCheckBox(String text)` - checkboxes are, by default, not selected
`JCheckBox(String text, boolean selected)`

Being a button, a JCheckBox can fire Action events. However, the preferred method of handling a JCheckBox is to use Item events. We can use the interface `ItemListener`, which has one method:

```
void itemStateChanged(ItemEvent e)
```

To get information about the event, we call the following methods on `ItemEvent` objects:

`Object getSource()`
`int getStateChange()` - returns `ItemEvent.ITEM_SELECTED` or `ItemEvent.ITEM_DESELECTED`
`Object getItem()` - this method usually returns a String containing the text of the item. We'll need to cast it into a String.

With this information, we can now understand the following example:

```
/* JCheckBoxDemo.java */

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

public class JCheckBoxDemo extends JApplet implements ItemListener {
      JCheckBox iBox, bBox;
      JLabel pretty;
      boolean italics, bold;

      public void init() {
            setLayout(new FlowLayout());

            iBox = new JCheckBox("Italics");
            iBox.setMnemonic(KeyEvent.VK_I);
            iBox.setToolTipText("Italicize the text");
            iBox.addItemListener(this);
            add(iBox);

            bBox = new JCheckBox("Bold");
            bBox.setMnemonic(KeyEvent.VK_B);
```

```
            bBox.setToolTipText("Bolden the text");
            bBox.addItemListener(this);
            add(bBox);

            pretty = new JLabel("Hello, World!");
            pretty.setFont(new Font("Serif", Font.PLAIN, 14));
            add(pretty);
      }

      public void itemStateChanged(ItemEvent e) {
            Object source = e.getSource();
            int state = e.getStateChange();
            if (source == iBox) {
                  if (state == ItemEvent.SELECTED) {
                        italics = true;
                  } else {
                        italics = false;
                  }
            } else if (source == bBox) {
                  if (state == ItemEvent.SELECTED) {
                        bold = true;
                  } else {
                        bold = false;
                  }
            }

            int mask = 0;
            if (italics) { mask |= Font.ITALIC; }
            if (bold) { mask |= Font.BOLD; }
            pretty.setFont(new Font("Serif", mask, 14));
            repaint();
      }
}
```

And as usual, here is our output on a 250x100 applet:

**Swing Components - Radio Buttons**

Another component that's derived from the AbstractButton is the JRadioButton. Similar to the check box, the radio button has all the same methods. In fact, even its constructors are identical. For convenience, they will be listed here anyway:

`JRadioButton(String text)` - radio buttons are, by default, not selected
`JRadioButton(String text, boolean selected)`

However, unlike check boxes, selecting a radio button will always deselect all other radio buttons in the group. However, before we can implement this feature, we must first figure out how to group buttons.

There is a class called ButtonGroup which does exactly this. Here are some of its methods:

`ButtonGroup()` - a constructor
`void add(AbstractButton button)` - note that you can add other buttons besides JRadioButtons. However, for some buttons, such as the JButton, it wouldn't make sense to do so. Note that this method implies the existence of other buttons that can be logically grouped.

As for working with radio buttons, every time a radio button is clicked, it generates an action event, an item event for the selected radio button (because clicking normally always selects a radio button), and usually a second item event for the deselected radio button. Handling radio buttons by action events or item events is good depending on the situation. However, generally, for efficiency, we want to use action events.

And now, for an example:

```
/* JRadioButtonDemo.java */

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

public class JRadioButtonDemo extends JApplet implements ActionListener
{
      ButtonGroup colors;
      JRadioButton rRed, rGreen, rBlue;
      JLabel colorful;

      public void init() {
            setLayout(new FlowLayout());

            colors = new ButtonGroup();

            rRed = new JRadioButton("Red");
            rRed.setMnemonic(KeyEvent.VK_R);
            rRed.setToolTipText("Paint the text red");
            rRed.addActionListener(this);
```

```
            colors.add(rRed);
            add(rRed);

            rGreen = new JRadioButton("Green");
            rGreen.setMnemonic(KeyEvent.VK_G);
            rGreen.setToolTipText("Paint the text green");
            rGreen.addActionListener(this);
            colors.add(rGreen);
            add(rGreen);

            rBlue = new JRadioButton("Blue");
            rBlue.setMnemonic(KeyEvent.VK_B);
            rBlue.setToolTipText("Paint the text blue");
            rBlue.addActionListener(this);
            colors.add(rBlue);
            add(rBlue);

            colorful = new JLabel("Hello, World!");
            add(colorful);
      }

      public void actionPerformed(ActionEvent e) {
            Object source = e.getSource();
            if (source == rRed) {
                  colorful.setForeground(Color.red);
            } else if (source == rGreen) {
                  colorful.setForeground(Color.green);
            } else if (source == rBlue) {
                  colorful.setForeground(Color.blue);
            }
            repaint();
      }
}
```

And here is our output in a 200x100 applet:

**Swing Components - Menus**

Now, no UI would be complete without a menu. Fortunately, our JApplet has the following methods to help us:

```
JMenuBar getJMenuBar()
void setJMenuBar(JMenuBar menubar)
```

The menu bar is that gray thing that runs across the top of our applications, for example the browser or Adobe Acrobat you're looking at my tutorial with. It's used to hold all the menus in a nice neat little row.

And JMenuBar has the following methods:

`JMenuBar()` - a constructor
`JMenu add(JMenu menu)` - note the return type; it returns the menu you just added

Now, for the JMenus:

`JMenu(String text)` - construct a menu with the given text
`JMenuItem add(JMenuItem menuitem)` - it returns the menu item you just added
`JMenuItem add(String text)` - a useful shortcut; it adds a menu item with the given text, then returns the `JMenuItem` object associated with that menu item
`void addSeparator()` - adds a separator (a little horizontal divider) to the bottom of the menu

Now, the thing about JMenuItems is that they are a subclass of AbstractButton, so they have most of the same methods other buttons have. Incidentally, JMenuItems work *exactly* the same as a JButton, with nearly all the same methods and with an identical constructor. You can set mnemonics and create tool tips, just like with a button. Here's the menu item constructor for reference.

```
JMenuItem(String text)
```

Incidentally, JMenu is a subclass of JMenuItem, so you can add a JMenu to a JMenu to create a sub-menu.

Also, there are two other subclasses of JMenuItem: JCheckBoxMenuItem and JRadioButtonMenuItem. These, too work *exactly* like their button counterparts, with many of the same methods and constructors. Here are the constructors for reference:

```
JCheckBox(String text)
JCheckBox(String text, boolean selected)
JRadioButton(String text)
JRadioButton(String text, boolean selected)
```

Now, we know enough to do a simple example:

```
/* JMenuDemo.java */

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

public class JMenuDemo extends JApplet implements ActionListener,
ItemListener {
      JMenuBar myMenuBar;
      JMenu options;
      JMenuItem nothing;
      JCheckBoxMenuItem iBox, bBox;
      boolean italics, bold;
      ButtonGroup colors;
      JRadioButtonMenuItem rRed, rGreen, rBlue;
      JLabel pretty;

      public void init() {
            setLayout(new FlowLayout());

            myMenuBar = new JMenuBar();
            setJMenuBar(myMenuBar);

            options = myMenuBar.add(new JMenu("Options"));
            options.setMnemonic(KeyEvent.VK_O);

            nothing = options.add("This does nothing");
            nothing.setMnemonic(KeyEvent.VK_T);

            options.addSeparator();

            iBox = new JCheckBoxMenuItem("Italic");
            iBox.setMnemonic(KeyEvent.VK_I);
            iBox.addItemListener(this);
            options.add(iBox);

            bBox = new JCheckBoxMenuItem("Bold");
            bBox.setMnemonic(KeyEvent.VK_B);
            bBox.addItemListener(this);
            options.add(bBox);

            options.addSeparator();

            colors = new ButtonGroup();

            rRed = new JRadioButtonMenuItem("Red");
            rRed.setMnemonic(KeyEvent.VK_R);
            rRed.addActionListener(this);
            colors.add(rRed);
            options.add(rRed);

            rGreen = new JRadioButtonMenuItem("Green");
            rGreen.setMnemonic(KeyEvent.VK_G);
            rGreen.addActionListener(this);
```

```
            colors.add(rGreen);
            options.add(rGreen);

            rBlue = new JRadioButtonMenuItem("Blue");
            rBlue.setMnemonic(KeyEvent.VK_U);
            rBlue.addActionListener(this);
            colors.add(rBlue);
            options.add(rBlue);

            pretty = new JLabel("Hello, World!");
            pretty.setFont(new Font("Serif", Font.PLAIN, 14));
            add(pretty);
      }

      public void itemStateChanged(ItemEvent e) {
            Object source = e.getSource();
            int state = e.getStateChange();
            if (source == iBox) {
                  if (state == ItemEvent.SELECTED) {
                        italics = true;
                  } else {
                        italics = false;
                  }
            } else if (source == bBox) {
                  if (state == ItemEvent.SELECTED) {
                        bold = true;
                  } else {
                        bold = false;
                  }
            }

            int mask = 0;
            if (italics) { mask |= Font.ITALIC; }
            if (bold) { mask |= Font.BOLD; }
            pretty.setFont(new Font("Serif", mask, 14));
            repaint();
      }

      public void actionPerformed(ActionEvent e) {
            Object source = e.getSource();
            if (source == rRed) {
                  pretty.setForeground(Color.red);
            } else if (source == rGreen) {
                  pretty.setForeground(Color.green);
            } else if (source == rBlue) {
                  pretty.setForeground(Color.blue);
            }
            repaint();
      }
}
```

And for our traditional screenshots (using a 250x150 applet):


Yes, the double menu does look kinda funny.

**Swing Components - JTabbedPane**

This is one of Swing's special space-saving components. This particular one allows us break our applet into multiple pages, each page accessed by a tab. Some of its useful methods include:

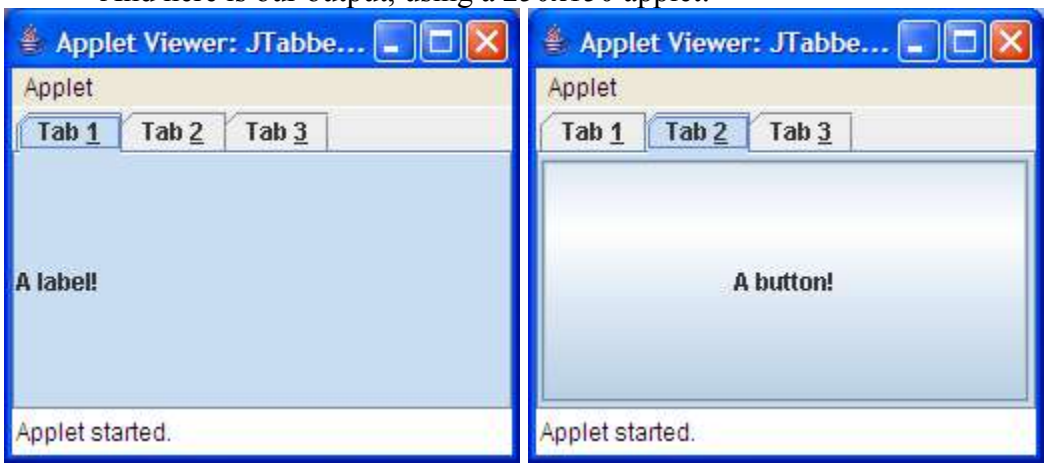
`JTabbedPane()` - a constructor
`JTabbedPane(int tabPlacement)` - use `JTabbedPane.TOP`, `JTabbedPane.BOTTOM`, `JTabbedPane.LEFT`, or `JTabbedPane.RIGHT`. Default placement is top.
`void addTab(String title, Component component)` - adds a tab with the given title and the given component
`int getMnemonicAt(int index)` - gets the mnemonic for the tab with the given index. Note that the first tab added has index 0, and the next tab added has index 1, and so forth.
`void setMnemonicAt(int index, int mnemonic)`
`String getToolTipTextAt(int index)`
`void setToolTipTextAt(int index, String text)`

And now, once again, it's example time:

```
/* JTabbedPaneDemo.java */

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

public class JTabbedPaneDemo extends JApplet {
      JTabbedPane myPane;
      JLabel ll;
      JButton bb;
```

```
	JCheckBox cb;

	public void init() {
		setLayout(new BorderLayout());

		myPane = new JTabbedPane();
		add(myPane);

		ll = new JLabel("A label!");
		myPane.addTab("Tab 1", ll);
		myPane.setMnemonicAt(0, KeyEvent.VK_1);

		bb = new JButton("A button!");
		bb.setToolTipText("This button does nothing.");
		myPane.addTab("Tab 2", bb);
		myPane.setMnemonicAt(1, KeyEvent.VK_2);

		cb = new JCheckBox("A check box!");
		cb.setToolTipText("This check box does nothing.");
		myPane.addTab("Tab 3", cb);
		myPane.setMnemonicAt(2, KeyEvent.VK_3);
	}
}
```

And here is our output, using a 250x150 applet:




Three things should jump out at you. The first one is the BorderLayout I used. This was just to make the JTabbedPane take up the whole applet. Try it using FlowLayout instead. You'll see it looks much uglier. I'll go over Layouts in the next tutorial.

The second thing is that you can only put one component per tab. And the third thing is that components (most notably the button) fill up the whole pane rather than being normal-sized. There's a way to solve both of these problems, but I'll cover it in the next tutorial where it comes naturally, but if you want a hint now: JPanel.

**Swing Components - the JFrame**

Now, to show you the most profound of all Swing components: the JFrame. This is profound because it allows you to break out of using applets for GUIs. A frame is a specific type of window--one used at the top-level and is generally never contained inside another window. (There is also the class JInternalFrame if you want to have sub-windows.)

To create a JFrame, we usually use five specific lines of code:

```
JFrame.setDefaultLookAndFeelDecorated(true);
```

Calling this on the static class JFrame ensures that our windows will be fully decorated.

```
JFrame fr = new JFrame("JFrameDemo");
```

This creates our frame. The argument passed to the constructor is the title to our window.

```
fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

This makes our program quit when our window is closed.

At this point, we add all our components to the frame, ie buttons and menus, using the `add()` method like in all the previous examples.

Next line:

```
fr.pack();
```

This formats the frame so that all our components appear correctly. An alternative would be to use `fr.setSize(int width, int height)`, but this may make some components appear correctly.

```
fr.setVisible(true);
```

This makes our frame appear. Our frame wouldn't be very useful if it were invisible, after all.

Frames and Applets are actually very closely related. In fact, they are so closely related that we could easily convert an applet program to a frame standalone application. Our next example is a conversion of our JButton example:

```
/* JFrameDemo.java */

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

public class JFrameDemo extends JFrame implements ActionListener {
      JButton bx, by;
```

```
	JLabel lx, ly;
	int countx, county;

	public JFrameDemo() {
		super("JFrameDemo");
	}

	void createGUI() {
		setLayout(new FlowLayout());

		bx = new JButton("Button X");
		bx.setMnemonic(KeyEvent.VK_X);
		bx.setToolTipText("This is a button");
		bx.addActionListener(this);
		add(bx);

		by = new JButton("Button Y");
		by.setMnemonic(KeyEvent.VK_Y);
		by.setToolTipText("This is a button");
		by.addActionListener(this);
		add(by);

		lx = new JLabel("Button X has been pushed 0 times");
		add(lx);

		ly = new JLabel("Button Y has been pushed 0 times");
		add(ly);
	}

	public void actionPerformed(ActionEvent e) {
		Object source = e.getSource();
		if (source == bx) {
			countx++;
			lx.setText("Button X has been pushed " + countx + "
times");
		} else if (source == by) {
			county++;
			ly.setText("Button Y has been pushed " + county + "
times");
		}
	}

	public static void main(String args[]) {
		JFrame.setDefaultLookAndFeelDecorated(true);
		JFrameDemo fr = new JFrameDemo();
		fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		fr.createGUI();
		fr.pack();
		fr.setVisible(true);
	}
}
```

Note that since this is a standalone application (hence the `main()` method), we can go back to using our traditional "`java JFrameDemo`". And this is what we get:


Hmm... I guess the `pack()` method wasn't so great after all. Now let's try doing `fr.setSize(250, 125)` instead:



*Much* better!

Note that there is also a class called `Frame`, the original AWT frame. Like the original AWT applet, it uses the simplified model without the root pane and content pane, so you could draw directly to it. It's very useful when you're making an application that only does painting and uses no components. When you create an AWT Frame, you don't use `setDefaultLookAndFeelDecorated()` nor `setDefaultCloseOperation()`, which simplifies your start up. However, this makes closing the window a bit more complicated. You'll have to use Window events and Window listeners to close the window and exit the program. We won't go over an AWT Frame example, as we're running low on time. However, you already have enough of the framework to figure out how to implement AWT Frames without much trouble.

Also note that the VidWorks project Demons (which is all painting and no components) uses the AWT Frame.

## Other Swing Components

There are many more Swing components out there. Here's a list that's not meant to be comprehensive, but only gives you a brief listing of the more commonly used Swing components:

**Top-Level Windows**
JDialog

**Organization Components**
JScrollPane
JSplitPane
JToolBar

**Complex Components**
JColorChooser
JFileChooser

**Control Components**
JComboBox
JList
JSlider
JSpinner
JToggleButton
JFormattedTextField
JPasswordField
JTextArea
JEditorPane
JTextPane
JTree
JTable

In addition, you can create your own custom components (although this would be a *very* advanced topic) by extending the `JComponent` class.

## Conclusion

This time, there won't be a unifying example, since quite frankly, this tutorial is big enough as is. However, we've shown you plenty of examples, and it should be pretty simple to combine them all to make a decent user interface, even if you haven't had *all* the concepts yet.

Next time: layout management, look & feel, and HTML formatting!

## Homework Assignment!

By use of combining Swing components, make a standalone program (ie, using a JFrame) that does something interesting when you click the buttons/menu items/other components. Creativity counts. Extra credit if you use some components that weren't covered in detail (like the ones listed in "Other Swing Components")!