GUI Programming in Java

Objectives

1. Explain the relationship between components and containers
2. Explain the difference between light-weight and heavy-weight components
3. Explain way it is unsafe to modify a GUI component on the application thread
4. Describe the pattern for event handling with components
5.

Introduction

Most end users expect to interact with a program through a graphical user interface (GUI). A GUI program has certain familiar features:

Window - A GUI program runs within a window. One window at a time has focus and receives input from the keyboard and mouse.

Menus - Commands the program accepts may be organized on menus, usually along the top of a window.

Components - Input and output can be through visual components such as buttons, labels, text fields, etc.

Support for writing GUI applications isn't normally part of a general purpose programming language. Support for writing GUI applications is usually provided by the operating system or a toolkit written for a particular operating system. Having a single source for UI components provides for consistency across applications running on the same machine.

However, some GUI support has to reside in the operating system. For example, the mouse and keyboard are connected directly to the operating system rather than a specific user program. Also, the window or area that a program occupies on the screen is known and partially managed by the operating system. The screen is a shared resource so it is managed by the operating system. So, a programming language or toolkit that provides support for writing GUI programs does so with primitive support from the operating system.

AWT or Abstract Window Toolkit is Java’s original toolkit for writing GUI programs. It's called abstract because Java only describes or implements the abstract interface to the toolkit. The specific implementation for the toolkit (the actual components displayed on the screen) is provided by the operating system. This is analogous to an abstract class defining an interface but a subclass providing the implementation.

AWT is a set of classes and interfaces that are part of the core Java API. In addition to AWT Java provides another set of interfaces and classes called swing which are also part of the Java API. Together AWT and swing provide support for writing platform independent GUI applications.
Given the close tie between a GUI toolkit and the operating system it's pretty amazing you can write a GUI program in Java and run it on multiple operating systems. What makes it work is a platform specific peer layer between Java's GUI toolkit and the native operating system. Peers are discussed in more detail when lightweight and heavyweight components are described.

GUI programming in general is challenging and the Java API for GUI programming isn't small so there is a lot to cover. In the first part of this lesson basic concepts are discussed along with the most common components. After completing this chapter you should be able to write a GUI program of medium complexity.

**Java GUI Packages**

Support for developing GUI programs in Java is spread across 4 packages:

- `java.awt.*`
- `java.awt.event.*`
- `javax.swing.*`
- `javax.swing.event.*`

(Remember package names may share the same prefix but there is no formal relationship between the two packages. For example, `java.awt` and `java.awt.event` share the same prefix and may contain similar elements, but you can't say the `java.awt` package contains the `java.awt.event` package.)

Originally all the support for writing GUI programs was contained in the AWT packages. The swing packages were added in Java 2. The swing packages are part of the Java Foundation Classes (JFC). The JFC all includes support for drag and drop and 2D graphics.

The swing packages build on AWT but also define some new components that supersede existing AWT components. For example, the component `javax.swing.JButton` supersedes AWT component `java.awt.Button`. You can still use superseded components in AWT (they aren't deprecated) but programmers are encouraged to move to the new swing components. Swing components are recommended over AWT components because swing components offer more features, are more efficient and are more consistent between platforms.

**Lightweight vs. Heavyweight Components**

AWT components which are displayed on the screen (buttons, check boxes, etc) come from the native operating system. If a window displays two AWT buttons the two buttons displayed are created and managed by the operating system:

Native operating system components are connected to a Java program through peer classes. AWT components are called heavyweight because for each component created three resources need to be created:
1. Java class representing the component
2. Java class representing the peer
3. Native operating system component

Most swing components are lightweight because they don't rely on a peer class or native operating system component. Only the top-level swing components (JFrame, JWindow, JApplet, and JDialog) are heavyweight or have a native implementation. Lightweight swing components draw themselves on the native window of the top-most container containing it (JFrame, Frame, etc). For lightweight swing components the only resource that needs to be created is:

1. Java class representing the component

As an example, if a javax.swing.JFrame contained a javax.swing.JButton the frame would be associated with a native operating system component, but the button doesn't need a peer or native operating system component. The button can draw itself on the frame it's displayed in. A swing button class needs to be a bit larger than an AWT button class because it has to know how to draw itself and react to user input, but this logic is much less than the resources required to create and display a native button.

Mixing AWT and Swing Components

You can mix AWT (heavyweight) and Swing (lightweight) components, but it's not recommended. It's especially a problem when you have overlapping AWT and Swing components. Heavyweight components always draw on top of lightweight components. This can be a problem, for example, if you have a lightweight menu component that needs to drop down over a heavyweight component in the application. Since heavyweight components are always displayed on top, the portion of the menu that overlaps the heavyweight component on the screen won't be visible.

For more information about mixing AWT and Swing components:

http://java.sun.com/products/jfc/tsc/articles/mixing/

Event-based programming

The high-level steps that most GUI programs follow are:

1. Create a window or top-level container
2. Create components and insert them into the top-level container
3. Wait for user interface events to arrive from components (for example, mouse click). Handle user interface events as they arrive.

Note, GUI programs follow a different paradigm. The paradigm is called event-based programming. Rather than your program having exclusive control flow your program sets up the user interface and waits for events to arrive. Events are directed to the proper event handling routines.

Containers

Containers group components. You can't just display a component on the screen. A component must be
placed in a container that is visible on the screen. JPanel is a swing container for grouping components. For example, here is a JPanel container that contains a textbox, a slider, and a combo box:

Note, the JPanel above also has a border defined for it.

Containers are themselves considered components so you can have a container within a container. For example, you might have a container with two components inside of it. The first component might be a button and the second another container with its own components. In implementation terms what it means that a container is a component is that the class that represents a container inherits from the class that represents a component. You can use a container anywhere you could use a component.

Top-level Containers

Some containers are special and can't be added to other containers. These special containers are called top-level containers. A top-level container defines the window in which the application will run. For example javax.swing.JFrame is a top-level container that defines a resizable window with a title bar and window buttons. Other top-level containers are JDialog, Dialog, Frame, and Applet.

Our first code example shows how to create a top-level frame for a GUI application.

```
import javax.swing.JFrame;

public class Spartan {
    // Create a frame, size it, and show it
    public static void main(String[] args) {
        JFrame frame = new JFrame("Spartan");
        frame.setSize(200,100);
        // show() will make the frame visible
        frame.show();
        System.out.println("Program begins...");
    }
}
```
A top-level window may also be an Applet, Window, or Dialog box.

Frequently used methods on JFrame include:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setSize(\text{int} \ width, \text{int} \ height)</td>
<td>Sets the width and height of the frame</td>
</tr>
<tr>
<td>pack()</td>
<td>Sets the width and height of the frame according to the preferred sizes of the components contained within the frame. You would call setSize() or pack() but not both.</td>
</tr>
<tr>
<td>show()</td>
<td>Make the window visible on the screen</td>
</tr>
<tr>
<td>isVisible(\text{boolean} \ b)</td>
<td>Does the same thing as show() when parameter b is true. Can be used to make a frame invisible</td>
</tr>
<tr>
<td>Container getContentPane()</td>
<td>Returns the container for this frame. You don't add components for display directly to the frame. Components you want to display on the frame are added to the frame's content pane.</td>
</tr>
<tr>
<td>setTitle(String title)</td>
<td>Set the title which will be displayed on the title bar. Note, you can also set the title with one of the JFrame constructors.</td>
</tr>
<tr>
<td>setLocation(\text{int} \ x, \text{int} \ y)</td>
<td>Used to position the frame on the screen. By default the frame is displayed with the upper left-hand corner of the frame at position 0,0 on the screen. Position 0,0 on the screen is the upper left-hand corner of the screen. (Note, in the example above the window appeared in the upper left-hand corner but was moved inside the DOS box for the screen capture.)</td>
</tr>
</tbody>
</table>

Unrelated to containers, the program above contains one other oddity. The main entry point finishes after "Program begins..." is printed but the application stays running? Why doesn't the application quit? What is keeping it running?

When you create a top-level UI component another "thread" (the event-dispatching thread or AWT thread) is created and started. Later we will have a complete lesson on threads and you will learn what they are and how to create them in your own programs. For now, here is a quick introduction to threads in general and the AWT thread in particular.

### Threads

A thread is a sequence of instructions to be executed. Every program has at least one thread of control.
When a Java program first begins the main thread of control starts with the main() entry point. When you create a top-level UI component another UI thread gets created for you. Why have another thread of control? You may take for granted the behavior of components. For example, when you click on a button it will look depressed (not sad but indented). You expect certain behavior of components that are displayed and since this behavior is the same for all components you don't expect to implement the behavior in your own programs. You expect the component to implement the behavior, possibly while your program is doing other work. This requires a separate active thread of control to handle UI features.

More importantly for the UI programs you will be writing, this separate thread of control is also responsible for notifying your program when an event occurs. Events are discussed in detail below. For now, here is a short program that demonstrates the existence of multiple threads of control

At this point you don't have to understand the complete program. The two important sections are highlighted with an arrow. The main entry point creates and makes visible a frame with one button and then goes into an infinite loop counting (with a 1 second pause between each number). Pressing the button causes an event which results in a call to actionPerformed() from the UI thread. The program demonstrates that there are two threads: the thread for the main entry point and a thread for the UI elements.

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

public class SwingThread extends JFrame implements ActionListener {

    private static int count = 0;

    public SwingThread(String title) {
        super(title);
        JButton incrementButton = new JButton("increment");
        incrementButton.addActionListener(this);

        Container contentPane = getContentPane();
        contentPane.add(incrementButton);
    }

    public void actionPerformed(ActionEvent evt) {
        count = 0;
    }

    // Main entry point
    public static void main(String[] args) {
        JFrame frame = new SwingThread("Swing Thread Demo");

        // Java idiom for handling window closing events
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });

        // pack() will resize the frame to its preferred size
        // which is calculated by adding up the preferred size
        // of the components within the frame.
        frame.pack();
    }
}
```
The increment button was pressed twice: 3 seconds after the application was started and then again 5 seconds later. If you haven't had any experience with multiple threads of control it may seem mysterious that two sections of code can appear to run simultaneously.

**Events**

Most UI programs first create and display components and then wait for events to be generated by user interaction. Events are generated for such interactions as:

- Window resize or window close
- Mouse move or mouse click
- Keypress
- Button press
- Text entry

Notice that events can be low level (mouse move, key press) or high-level (button press, text entry). High-level events are called semantic events. It's usually better to listen for semantic events rather than low level events. For example, if you just check for mouse clicks over a button you will not register a button press that is generated by a keyboard short-cut.

Events originate in components and are sent to all registered listeners. A callback mechanism is used to notify registered listeners when an event occurs. The following (pseudo UML) diagram shows the sequence of steps for registering a listener with a JButton and receiving a callback later. The listener is interested in button press events.
The UI program creates a JButton and registers itself as interested in button press events that originate in the button. When the user presses the button an ActionEvent is created that contains information about the button press. This event is sent to all registered listeners.

Here is the complete program:

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

public class ButtonEvent extends JFrame implements ActionListener {
  public ButtonEvent() {
    // Set the title of the window
    super("Action Event Example");

    JButton b = new JButton("Press for Action");
    b.addActionListener(this);

    // Components must be added to the
    // content pane of a JFrame
    Container contentPane = getContentPane();
    contentPane.add(b);
  }

  public void actionPerformed(ActionEvent evt) {
    System.out.println("Button Pressed");
  }

  // Main entry point
  public static void main(String[] args) {
    JFrame frame = new ButtonEvent();

    // pack() will resize the frame to its preferred size
    // which is calculated by adding up the preferred size
    // of the components within the frame.
    frame.pack();
    // show() will make the frame visible
    frame.show();
  }
}
```
Does the program about have multiple threads of control? (No, the main thread of control ends after the frame is displayed. All work is performed in the event handling routine on the UI thread.)

The method actionPerformed() is called an event handling routine. It is called from the UI thread.

The consequences of event handling routines executing on the UI thread are:

1. Your event handler routines should be as efficient as possible. While your event handler routine is executing no other UI activity can take place. If your event handler routine can't finish quickly you can always create and use a new thread to perform the work.

2. If your program maintains a separate thread of control or creates a thread to execute time consuming event handler routines you may need to synchronize thread activity. If multiple threads need simultaneous access to data you will need to synchronize access. Multithreaded programming is discussed in a separate lesson.

---

### Extra for Experts

Swing is not thread safe. This means that methods on swing components don't synchronize their access to common data in the swing library. Swing components don't expect to be called by any thread other than the UI thread. (You can, however, safely call swing methods from another thread before they are visible on the screen. The UI thread never calls a component before it is visible on the screen.) If you need to modify swing elements already visible you should do so on the UI thread (also called the AWT event dispatching thread). There are two options: (1) modify swing elements with event handling routines because event handling routines are called by the AWT event dispatching thread. Or, (2) schedule a swing event to perform the update. Here is how you would schedule a swing event to perform the update:

```java
Runnable someAction = new Runnable() {
    public void run() {
        System.out.println("AWT thread: " + Thread.currentThread());
    }
};
SwingUtilities.invokeLater(someAction);
System.out.println("My thread: " + Thread.currentThread());
```

The example above shows how to listen for and handle button press events (ActionEvents). There are different types of events for different types of components. For example, window components will emit Window events.

There is a pattern to event handling: [tbd: add a picture here that shows the relationship between all these elements.]
1. The event is named \texttt{xxEvent} (ie \texttt{ActionEvent})

2. Components that generate events of type \texttt{xxEvent} have a method called \texttt{add\texttt{xx}Listener} (ie \texttt{addActionListener}). This method is used to register a reference to a class that wants to be notified when events of type \texttt{xxEvent} occur.

3. An interface defines the methods that will be used to deliver \texttt{xxEvents}. The interface is called \texttt{xxListener} (ie \texttt{ActionListener}). Classes interested in receiving events of type \texttt{xxEvent} implement the interface and register themselves with the component.

4. The methods which deliver events of type \texttt{xxEvent} have names with the pattern \texttt{xx<verb>} (ie \texttt{actionPerformed})

5. If the interface defines more than a few methods an adapter class is provided. (\texttt{ActionListener} doesn't have an adapter because it has only one method. \texttt{WindowAdapter} is an example adapter class.) An adapter class is a convenience class provided for interfaces with more than a few methods.

An adapter class is a convenience class that provides an empty implementation for all of the methods of an interface. Rather than implement all methods in an interface you can extend its adapter class and override only the methods you are interested in. For example, here is the adapter class for \texttt{WindowEvents}:

```java
public abstract class WindowAdapter implements WindowListener {
    public void windowOpened(WindowEvent e) {}
    public void windowClosing(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
}
```
Why is WindowAdapter declared to be abstract? It doesn't have to be because it provides implementation (albeit a null implementation) for all of the methods in WindowListener. Declaring it abstract is a design decision. It says that this class was designed to be extended rather than used alone.

Here are the event classes and related methods for the most popular events:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Adapter Class</th>
<th>Add Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface ActionListener {}</td>
<td>(none)</td>
<td>AbstractButton::addActionListener()</td>
</tr>
<tr>
<td>interface MouseListener {}</td>
<td>MouseAdapter</td>
<td>Component::addMouseListener()</td>
</tr>
<tr>
<td>interface MouseMotionListener</td>
<td>MouseMotionAdapter</td>
<td>Component::addMouseMotionListener()</td>
</tr>
<tr>
<td>interface WindowListener {}</td>
<td>WindowAdapter</td>
<td>Window::addWindowListener()</td>
</tr>
<tr>
<td>interface ItemListener {}</td>
<td>(none)</td>
<td>AbstractButton::addItemListener()</td>
</tr>
</tbody>
</table>
Here are the semantics for each of the event types above:

<table>
<thead>
<tr>
<th>Event</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionEvent</td>
<td>Generated by pressing a button and pressing enter at a text fields</td>
</tr>
<tr>
<td>MouseEvent</td>
<td>Used for mouse movements (MouseMotionListener) and mouse button press events (MouseListener)</td>
</tr>
<tr>
<td>WindowEvent</td>
<td>Generated by a window</td>
</tr>
<tr>
<td>ItemEvent</td>
<td>Generated by check boxes and radio buttons</td>
</tr>
</tbody>
</table>

When the same event handler is used for multiple components you need to check the source of the event. Here is an example:

**Simple Example**

This example also demonstrates how containers can be used to group components. Grouping components with containers is discussed in more detail later.

The following program displays two JButtons in a JPanel in a top-level JFrame:

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

public class ButtonExample extends JFrame implements ActionListener {

    private JButton b1 = null;
    private JButton b2 = null;

    public ButtonExample() {
        // Set the title of the window
        super("Button Example");

        // We want the buttons to display at their preferred size, so we can't use the default JFrame layout
        // BorderLayout. Here we switch it to FlowLayout.
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        b1 = new JButton("Button #1");
        b1.addActionListener(this);

        b2 = new JButton("Button #2");
        b2.addActionListener(this);

        contentPane.add(b1);
    }
}
```

This program creates two buttons and adds them to the content pane of a JFrame. When the buttons are clicked, an action listener is triggered, demonstrating how event handling can be used in GUI programming.
// This event handler routine is shared by two
// components so we need to query the event
// to find out which component the event
// is coming from.

public void actionPerformed(ActionEvent evt) {
    if (evt.getSource() == b1) {
        System.out.println("Button #1 pressed");
    }
    else if (evt.getSource() == b2) {
        System.out.println("Button #2 pressed");
    }
    else {
        System.out.println("Unexpected event: " + evt);
    }
}

// Main entry point

public static void main(String[] args) {
    JFrame frame = new ButtonExample();

    // Java idiom for registering a handler for
    // a window close event.
    frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });

    // pack() will resize the frame to its preferred size
    // which is calculated by adding up the preferred size
    // of the components within the frame.
    frame.pack();

    // You also might see programs that call show()
    // to make the frame visible on the screen. show() and
    // setVisible(true) are equivalent.
    frame.setVisible(true);
}

Handling events with an anonymous inner classes

Note, the program above uses an anonymous inner class to listen for window closing events. It could also have been specified:

```java
WindowListener l = new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
};
frame.addWindowListener(l);
```

Is the above still anonymous and inner? (The answer is yes.)

Here is the same functionality but it is not anonymous and not inner.
public static void main(String[] args) {
    WindowListener wl = new MyWindowListener();
    frame.addWindowListener(wl);
}

class MyWindowListener extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}

The pattern is:

reference = new <class> {
    public void method-in-class-to-overide() { . . .}  
    // Why will it always be an overridden method?
    // why wouldn't you ever have any new methods here?
    // Answer: It's an anonymous class. You can only call
    // methods on this class from references to one
    // of its super classes.
};

OR,
reference = new <interface> {
    public void definition-for-interface-method() { . . .}
    // you must provide implementation for all methods
    // defined in the interface
};

We could also have used anonymous inner classes to handle the button events:

public ButtonExample() {
    b1 = new JButton("Button #1");
    b1.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
            System.out.println("Button #1 pressed");
        }
    });

    b2 = new JButton("Button #2");
    b2.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
            System.out.println("Button #2 pressed");
        }
    });

    add(b1);
    add(b2);
}

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == b1) {
            System.out.println("Button #1 pressed");
        } else {
            System.out.println("Button #2 pressed");
        }
    }
Anonymous inner classes are intimidating at first but do make it easier to specify an event handling routine and to keep the routine close to the data structures it refers to.

**Mouse Events**

Here is an example of an application that handles mouse events. The program works like an etch-a-sketch:

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

public class Scribble {
    // Main entry point
    public static void main(String[] args) {
        JFrame frame = new DrawFrame();
        // The window size is set in the constructor for DrawFrame
        frame.show();
    }
}

class DrawFrame extends JFrame {
    public DrawFrame() {
        setTitle("Scribble");
        setSize(600, 400);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        // Components that will appear on a JFrame
        // are added to the content pane.
        Container contentPane = getContentPane();
        DisplayArea da = new DisplayArea();
        contentPane.add(da, BorderLayout.CENTER);
    }
}
```
A JPanel is a component you can draw on. Our panel is listening for mouse events.

class DisplayArea extends JPanel implements MouseMotionListener, MouseListener {

    public DisplayArea() {
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    // --------------------------------------------------------
    // There are two methods in the MouseMotionListener (move, drag) interface.
    public void mouseMoved(MouseEvent evt) {
    }

    public void mouseDragged(MouseEvent evt) {
        int new_x = evt.getX();
        int new_y = evt.getY();
        Point old = (Point)currentLine.lastElement();
        Graphics g = getGraphics();
        g.drawLine(old.x, old.y, new_x, new_y);
        currentLine.addElement(new Point(new_x, new_y));
    }

    // --------------------------------------------------------
    // There are 5 methods in the MouseListener (click, double click, etc) interface.
    public void mousePressed(MouseEvent evt) {
        int x = evt.getX();
        int y = evt.getY();
        currentLine = new Vector();
        currentLine.addElement(new Point(x, y));
    }

    public void mouseReleased(MouseEvent evt) {
        int x = evt.getX();
        int y = evt.getY();
        currentLine.addElement(new Point(x, y));
        if (onScreenLines == null)
            onScreenLines = new Vector();
        onScreenLines.add(currentLine);
    }

    // We don't process these events but must declare the methods here because we are obliged to implement all methods in the interface.
    public void mouseClicked(MouseEvent evt) {
    }

    public void mouseEntered(MouseEvent evt) {
    }
}

GUI Programming in Java http://jpartyka.com/backup/java/binBackup/Java tutorials/GUI Progra...
public void mouseExited(MouseEvent evt) {
    
    // This method is responsible for painting the component.
    public void paintComponent(Graphics g) {
        // This will repaint the background
        super.paintComponent(g);

        if (onScreenLines == null)
            return;

        for (Enumeration e = onScreenLines.elements() ;
            e.hasMoreElements() ;)
        {
            Vector lines = (Vector)e.nextElement();

            Enumeration e2 = lines.elements();
            Point start = (Point)e2.nextElement();
            for (; e2.hasMoreElements() ;)
            {
                Point end = (Point)e2.nextElement();
                g.drawLine(start.x,start.y,end.x,end.y);
                start = end;
            }
        }
    }

    private Vector onScreenLines = null;
    private Vector currentLine = null;
}

The example above also demonstrates how to draw directly on the window. A JPanel is both a container and the preferred base component for drawing directly to the screen. The ability to draw directly to the screen allows you to create custom components.

The method responsible for drawing or painting a component is:

public void paintComponent(Graphics g)

paintComponent() is a method of JComponent.

If you want to draw directly to an area of the screen you would create a JPanel and override its paintComponent() method.

paintComponent is called by the UI thread when it has been determined that the component needs to repaint itself. A request to repaint may originate from:

1. The operating system or UI thread (for example when the window is resized or made visible after being iconified)

2. Your program (for example if the data the component represents has changed you will want to request that the display be refreshed)

You can override the paintComponent() method for any component, but you would be interfering with the normal logic for painting the component. For example, the following program overrides the paintComponent() method of JButton.
import javax.swing.event.*;

public class CustomButton extends JButton {

    public CustomButton(String label) {
        super(label);
    }

    public void paintComponent(Graphics g) {
        // Draw the button as it normally would look
        super.paintComponent(g);

        // Add some graffiti
        g.drawString("Buttons are bad!",5,14);
    }

    // Main entry point
    public static void main(String[] args) {
        JFrame frame = new JFrame("Custom Button");

        JButton customButton = new CustomButton("Press Here");

        Container contentPane = frame.getContentPane();
        contentPane.add(customButton);

        // pack() will resize the frame to its preferred size
        // which is calculated by adding up the preferred size
        // of the components within the frame.
        frame.pack();

        // show() will make the frame visible
        frame.show();
    }
}

Notice that your program should never call paintComponent() directly. To repaint a component your program should call its repaint() method. repaint() will schedule a call to paintComponent() on the AWT event dispatching thread. Painting must be done at the request of the UI thread. One reason is that the UI thread creates the graphics context (Graphics) through which the window is painted.

JDK 1.0 AWT

Painting was done differently with AWT 1.0 components. You may see code written for AWT components (or AWT code incorrectly applied to swing components) so here is a quick primer on painting as it was done with AWT components in Java 1.0.

Before JPanel was added custom components inherited from the Canvas class and overrode the paint() method to draw directly to a window. The other methods used were:
• repaint() - used to schedule a screen repaint (used for the same thing today).

• update() - repainted the background and called paint(). Often overrode to prevent the screen from flickering.

• paint() - used to draw directly to the window

Right and Left Mouse Buttons

The mouse listener methods don't distinguish between a right and left mouse click. Detecting right and left mouse clicks isn't an integral part of the API. This isn't unreasonable since right and left mouse clicks are not supported on all platforms. For example, the mouse on a Macintosh computer has only one mouse button. You can, however, distinguish between right and left mouse clicks by checking flags or modifiers on the mouse event. Here is an example:

```java
public void mousePressed(MouseEvent evt) {
    if ((evt.getModifiers() & InputEvent.BUTTON1_MASK) != 0)
        System.out.println("Left mouse click");
    if ((evt.getModifiers() & InputEvent.BUTTON2_MASK) != 0)
        System.out.println("Middle mouse click");
    if ((evt.getModifiers() & InputEvent.BUTTON3_MASK) != 0)
        System.out.println("Right mouse click");
}
```

Layout Managers

Containers hold components. The size and location of components within a container are determined by the container's layout manager. Layout isn't controlled by the container, it's controlled by the layout manager associated with the container.

Each container has associated with it a layout manager. The layout manager is responsible for determining the size and location of the components in the container. Why not just place each component at a specific position in the window? You can set the layout manager to null and take responsibility for positioning and sizing all of the components of the container yourself. However, this makes your application less portable. A layout manager will help ensure your application looks good on monitors of different sizes and resolutions.

Containers have default layouts. The default layout manager for the content pane of a JFrame is BorderLayout. The default layout for a JPanel is FlowLayout.

We will discuss 2 layout managers in this class: BorderLayout and FlowLayout.

Two main issues related to layout managers:

1. If you aren't using the default layout manager you need to set a specific layout manager, and

2. When you add components you must consider the layout manager in effect. For example, some layout managers require special parameters. Some may require elements to be added in a certain order.
Border Layout

The BorderLayout has 5 different areas: North, South, East, West, Center. Here is an example, that demonstrates the border layout:

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

public class BorderLayoutExample extends JFrame {

    public BorderLayoutExample() {
        // The default layout for the content pane
        // of a JFrame is BorderLayout
        Container contentPane = getContentPane();

        JButton b1 = new JButton("North");
        // If you don't specify the direction
        // as part of the add method the default
        // is "Center",
        contentPane.add(b1,BorderLayout.NORTH);

        JButton b2 = new JButton("South");
        contentPane.add(b2,BorderLayout.SOUTH);

        JButton b3 = new JButton("East");
        contentPane.add(b3,BorderLayout.EAST);

        JButton b4 = new JButton("West");
        contentPane.add(b4,BorderLayout.WEST);

        JButton b5 = new JButton("Center");
        contentPane.add(b5,BorderLayout.CENTER);
    }

    public static void main(String[] args) {
        JFrame frame = new BorderLayoutExample();
        frame.pack();
        frame.show();
    }
}
```

The program above produces a window that looks like:

Notice that the layout handles both positioning and size. The border layout is one that resizes all of the components to fit the available room. (Other layout managers will let components remain at their "preferred" size.) The border layout stretches the center left/right top/bottom to fill the available space.
The north and south buttons are stretched left/right to fill the available space. The east west button are stretched top/bottom to fill up the available space. So, the north and south components remain at their preferred height and the east and west components remain at their preferred width.

Flow Layout

Another useful layout is flow layout. Here is the example above using a FlowLayout rather than the default border layout:

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

public class FlowLayoutExample extends JFrame {

    public FlowLayoutExample() {
        // The default layout is BorderLayout so
        // we need to change the layout to Flow
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        JButton b1 = new JButton("North");
        // The flow layout manager doesn't require
        // a second parameter to the add method.
        // components are layed out left to right
        // according to the order they were added
        // and remain at their preferred size.
        contentPane.add(b1);

        JButton b2 = new JButton("South");
        contentPane.add(b2);

        JButton b3 = new JButton("East");
        contentPane.add(b3);

        JButton b4 = new JButton("West");
        contentPane.add(b4);

        JButton b5 = new JButton("Center");
        contentPane.add(b5);
    }

    public static void main(String[] args) {
        JFrame frame = new FlowLayoutExample();
        frame.pack();
        frame.show();
    }
}
```

A JPanel is a container class you can use to arrange components on the screen. The default layout manager for a JPanel is flow layout.
In the example below the JFrame container contains two components. In the center it has a button component. On the south side it has a JPanel or container component. Inside the JPanel there are two buttons.

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

public class FlowLayoutExample extends JFrame {
    public FlowLayoutExample() {
        Container contentPane = getContentPane();
        JButton b1 = new JButton("Big Center Button");
        contentPane.add(b1, BorderLayout.CENTER);

        // The default layout for a JPanel is FlowLayout
        JPanel tempPanel = new JPanel();
        JButton b2 = new JButton("West");
        JButton b3 = new JButton("East");
        tempPanel.add(b2);
        tempPanel.add(b3);

        // Now add the container to the container
        contentPane.add(tempPanel, BorderLayout.SOUTH);
    }

    public static void main(String[] args) {
        JFrame frame = new FlowLayoutExample();
        frame.pack();
        frame.show();
    }
}
```

Notice that the flow layout doesn’t try to resize the components to fit the available area. The FlowLayout manager lets components display at their preferred size.

With these two layout managers (flow and border) and the ability to nest containers within containers you can create moderately sophisticated user interfaces.

**Grid Layout**

The next example departs from the style of the UI programs above but has most of the same elements.
public class GridLayoutExample {

    public static void main(String[] args) {
        JFrame frame = new JFrame("Grid Layout Example");

        Container contentPane = frame.getContentPane();
        contentPane.setLayout(new GridLayout(3, 2));

        JButton b = new JButton("1");
        contentPane.add(b);

        b = new JButton("2");
        contentPane.add(b);

        b = new JButton("3");
        contentPane.add(b);

        b = new JButton("4");
        contentPane.add(b);

        b = new JButton("5");
        contentPane.add(b);

        b = new JButton("6");
        contentPane.add(b);

        frame.pack();
        frame.show();
    }
}

Here is the output of the above program. Notice, that the grid layout divides the available space evenly between components.

Grid Bag Layout

The most powerful (and most feared) layout of all is the GridBagLayout.

Here is a popular pattern for using the GridBagLayout. This code was used to create the following UI:
The 5 most popular layout managers are: BorderLayout, FlowLayout, GridBagLayout, BoxLayout, and GridLayout. BoxLayout is a vertical or horizontal layout and GridLayout is a fixed size.

**JTextArea**

A JTextArea is a multi-line text component. Some of the features it provides are:

1. It is scrollable (although it doesn't implement scrolling)
2. It can be configured for line/word wrap
3. It can be editable or non-editable

Here is a simple example. The application looks like:

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

public class TextAreaExample extends JFrame implements ActionListener {

    private JTextArea sourcePane = null;
    private JButton lockButton = null;
    private JButton clearButton = null;
    private boolean isLocked = false;

    public TextAreaExample() {
        Container contentPane = getContentPane();
        sourcePane = new JTextArea(10,20);
        // By default a text area doesn't provide
        // line wrap.
        sourcePane.setLineWrap(true);
        // By default a text area wraps at characters
        // this will cause it to wrap words.
        sourcePane.setWrapStyleWord(true);
        // A JTextArea doesn't implement scrolling
        // JTextArea implements the Scrollable interface
        // which allows it to be scrolled by another
        // class.
        JScrollPane sourceScrollPane = new JScrollPane(sourcePane);
        contentPane.add(sourceScrollPane,BorderLayout.CENTER);
        // Buttons are grouped in a panel at
        // the bottom of the window.
        // Putting the buttons inside a JPanel
        // allows them to display at their
        // preferred sizes.
        JPanel buttonPanel = new JPanel();
        lockButton = new JButton("Lock");
        lockButton.addActionListener(this);
        buttonPanel.add(lockButton);

        clearButton = new JButton("Clear");
        clearButton.addActionListener(this);
        buttonPanel.add(clearButton);

        // Add the button panel at the bottom of the JFrame
        contentPane.add(buttonPanel,BorderLayout.SOUTH);
    }

    public void actionPerformed(ActionEvent evt) {
        if (evt.getSource() == lockButton) {
            isLocked = !isLocked;
            if (isLocked) {
                lockButton.setText("Unlock");
                sourcePane.setEditable(false);
            }
        } else {
            lockButton.setText("Lock");
        }
    }
}
sourcePane.setEditable(true);

} else if (evt.getSource() == clearButton) {
    sourcePane.setText(null);
}

public static void main(String[] args) {
    JFrame frame = new TextAreaExample();
    frame.setTitle("Text Area Example");

    // Anonymous Inner Class
    WindowListener l = new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    }

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

If you are adding text to the end of a JTextArea and want the display to scroll automatically as you add text so that the last few lines of the text field are always visible:

sourcePane.append(newMessage);
sourcePane.setCaretPosition(sourcePane.getDocument().getLength());

Borders

JComponents can have borders. The border of a component is the area that surrounds the perimeter of the component. Borders are useful for outlining components or adding space around a component.

For example, here is a JLabel with a titled border:

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

// class Border is in a separate swing package
import javax.swing.border.);

public class SimpleBorder extends JFrame {
    public SimpleBorder() {
        JLabel label = new JLabel("Label Value",JLabel.CENTER);
```

26 of 40 24/3/2012 3:20 PM
Border titledBorder = BorderFactory.createTitledBorder("The Title");
label.setBorder(titledBorder);

Container contentPane = getContentPane();
contentPane.add(label);

public static void main(String[] args) {
    JFrame frame = new SimpleBorder();
    frame.pack();
    frame.setVisible(true);
}

The BorderFactory class provides static convenience methods for creating borders. Other useful borders are:

Lowered Bevel Border

Border bevelBorder = BorderFactory.createLoweredBevelBorder();

Etched Border

Border etchedBorder = BorderFactory.createEtchedBorder();

Empty space

// Parameters are room in pixels (top,left,bottom,right)
Border emptyBorder = BorderFactory.createEmptyBorder(20,10,5,5);

Adding borders to the Text Area example given above:
JScrollPane sourceScrollPane = new JScrollPane(sourcePane);
// Some swing components already have a border. Rather than
// replace the border around a scroll pane we will replace
// the border around the scroll pane with a compound border

Border titledBorder = BorderFactory.createTitledBorder("TextArea");
// Order is (outside border, inside border)
Border compoundBorder = BorderFactory.createCompoundBorder(titledBorder,
sourceScrollPane.getBorder());

sourceScrollPane.setBorder(compoundBorder);

JPanel buttonPanel = new JPanel();
// Parameters specify space (pixels)
// in the order: top, left, bottom, right
Border emptyBorder = BorderFactory.createEmptyBorder(5,5,5,5);
buttonPanel.setBorder(emptyBorder);

**Menus**

Menus are a convenient way to provide features in a way that doesn't take up valuable screen
real-estate.

You can add a menu bar (JMenuBar) to a JApplet, JDialog, JFrame, JInternalFrame or JRootPane.

A menu bar contains menus (JMenu). A menu contains menu items (JMenuItem). Because a JMenu is
a JMenuItem you can have nested menus.

Menu items behave like buttons. Menu items descend from JAbstractButton.

![MenuTypeInfo](image.png)

Adding a menu to our Text Area example:
// Add the button panel at the bottom of the JFrame
contentPane.add(buttonPanel,BorderLayout.SOUTH);

// Add a menu

// We can set the menu bar and then add menus to it
JMenuBar menuBar = new JMenuBar();
setJMenuBar(menuBar);

// Create the first menu.
JMenu menu = new JMenu("File");
menu.setMnemonic(KeyEvent.VK_F);
menuBar.add(menu);

// Create an item for the first menu
exitMenuItem = new JMenuItem("Exit",KeyEvent.VK_X);
exitMenuItem.addActionListener(this);

//menuItem.setMnemonic(KeyEvent.VK_X); // set in constructor
menu.add(exitMenuItem);

public void actionPerformed(ActionEvent evt) {
    if (evt.getSource() == lockButton) {
        isLocked = !isLocked;
        if (isLocked) {
            lockButton.setText("Unlock");
            sourcePane.setEditable(false);
        }
        else {
            lockButton.setText("Lock");
            sourcePane.setEditable(true);
        }
    }
    else if (evt.getSource() == clearButton) {
        sourcePane.setText(null);
    }
    else if (evt.getSource() == exitMenuItem) {
        }
Dialog Boxes

Dialog boxes are a convenient way of organizing related input and output of a GUI application. A dialog box is a more limited form of a stand-alone window or frame.

Typically dialog boxes are initiated from menu commands. A menu command that end in ... brings up a dialog box:

Dialog boxes have many of the features common to windows and frames:

Dialog boxes are separate moveable windows with a title bar and close icon. Dialog boxes present a portion of the state of an application and often take input that affects the state of the application.

Most dialog boxes are modal. Modal means that while the dialog box has focus other windows of the application are locked.

Adding Dialog Boxes to a Program
There are a variety of options for adding dialog boxes to an application. For specific tasks such as selecting a file or picking a color there are predefined dialog boxes (JFileChooser, and JColorChooser). For simple messages and single-value inputs there is JOptionPane. JOptionPane is a convenience class that makes it easy to create simple standard dialog boxes. You can also create custom multi-input dialog boxes with JOptionPane. For maximum flexibility there is JDialog. The convenience class JOptionPane creates dialog boxes using JDialog. If you need maximum flexibility you can work with JDialog directly. However, if you create a dialog box directly from JDialog many of the details that JOptionPane handles are now your responsibility.

JFileChooser

[tbd]

Simple Dialog Boxes with JOptionPane

The book gives examples of how to use JOptionPane for simple single-value inputs. There is also a good online tutorial at the Sun web site that explains how to use JOptionPane.

Custom Dialog Boxes with JOptionPane

JOptionPane has several overloaded static methods called showInputDialog() that make it easy to read a single value from the user. If you need a multi-input value dialog box and don't need the full flexibility (or want the full responsibility) of JDialog box your best option might be the overloaded static methods called showOptionDialog() of JOptionPane. These methods allow you to create arbitrarily complex dialog boxes without dealing with all of the details that JDialog box exposes.

This next example shows how to use the showOptionDialog() feature of JOptionPane. It also demonstrates a convention for managing the state of an application that is configurable though a dialog box.

The example displays a formatted label at the center of the window. A dialog box tied to a menu command is used to change the font properties of the label.

The only two font properties that are modifiable are style (bold or plain) and size.
Selecting OK applies any changes made to the message.

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

public class DialogDemo extends JFrame {

    // model represents the data that
    // the dialog box will display to
    // the user and get from the user
    private FontModel model = null;
    private JLabel message = null;

    public DialogDemo() {
        super("Dialog Demo");

        // model contains the data that
        // the font dialog box will
        // display and get
        model = new FontModel();

        // JLabel implements interface SwingConstants which
        // defines constants LEFT, CENTER
        message = new JLabel("Hello World!", JLabel.CENTER);

        updateFont(model);
        Container contentPane = getContentPane();
    }
}
```
contentPane.add(message);

// Add menu
JMenuBar menuBar;
JMenu menu;
JMenuItem menuItem;

menuBar = new JMenuBar();
setJMenuBar(menuBar);

// Build the first menu.
menu = new JMenu("File");
menu.setMnemonic(KeyEvent.VK_F);
menuBar.add(menu);

menuItem = new JMenuItem("Exit", KeyEvent.VK_X);
menuItem.setMnemonic(KeyEvent.VK_X);
menuItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        System.exit(0);
    }
});
menu.add(menuItem);

// Build the second menu.
menu = new JMenu("Format");
menu.setMnemonic(KeyEvent.VK_F);
menuBar.add(menu);

menuItem = new JMenuItem("Font...", KeyEvent.VK_F);
menuItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        model = FontChooser.showDialog(DialogDemo.this, model);
        // If the model reference returned is the same
        // as the one passed (user pressed cancel)
        // we could skip the following update.
        updateFont(model);
    }
});
menu.add(menuItem);

// This menu has two items
menuItem = new JMenuItem("Paragraph...", KeyEvent.VK_P);
menuItem.setEnabled(false); // Not used in this application
menuBar.add(menuItem);

private void updateFont(FontModel model) {
    int fontStyle;
    if (model.isBold())
        fontStyle = Font.BOLD;
    else
        fontStyle = Font.PLAIN;
    Font f = new Font("SansSerif", fontStyle, model.getSize());
    message.setFont(f);
}

public static void main(String[] args) {
JFrame frame = new DialogDemo();

frame.addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});

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

class FontModel implements Cloneable {
    private boolean bold;
    private int size;

    public FontModel() {
        bold = false;
        size = 12;
    }

    public boolean isBold() {
        return bold;
    }

    public int getSize() {
        return size;
    }

    public void setBold(boolean b) {
        bold = b;
    }

    public void setSize(int s) {
        size = s;
    }

    // clone() in Object is protected
    // we override clone() here and
    // make it public so clients
    // can close instances of this class
    public Object clone() {
        FontModel newFontModel;

        try {
            newFontModel = (FontModel) super.clone();
            // No special clone processing since
            // all attributes are primitive.
        } catch (CloneNotSupportedException e) {
            // Should never happen
            e.printStackTrace();
            newFontModel = null;
        }

        return newFontModel;
    }
}
class FontChooser extends JPanel {

private JCheckBox boldCheckBox;
private JComboBox fontSizeComboBox;
private FontModel model;

public FontChooser(FontModel model) {
    this.model = model;

    setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
    setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

    boldCheckBox = new JCheckBox("Bold");
    boldCheckBox.setSelected(model.isBold());

    String opsString[] = {"8","10","12","14","28");
    fontSizeComboBox = new JComboBox(opsString);
    fontSizeComboBox.setSelectedItem(Integer.toString(model.getSize()));

    add(boldCheckBox);
    add(fontSizeComboBox);
}

// Before we return the model we do any pending lazy updates
// The model may change dynamically during each UI action
// or some model updates may be postponed until this point.
private FontModel getModel() {
model.setSize(Integer.parseInt((String)fontSizeComboBox.getSelectedItem()));
model.setBold(boldCheckBox.isSelected());
return model;
}

public static FontModel showDialog(Component parent, FontModel model) {
FontModel newModel = (FontModel)model.clone();
FontChooser fontChooser = new FontChooser(newModel);

Object[] options = {"OK", "Cancel"};

int n = JOptionPane.showOptionDialog(parent,
fontChooser,
"Font Dialog",
JOptionPane.YES_NO_OPTION,
JOptionPane.PLAIN_MESSAGE,
null,
options,
options[0]
);

if (n == JOptionPane.YES_OPTION) {
    return fontChooser.getModel();
} else
    return model;
}

Custom Dialog Boxes with JDialog
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;

public class DialogDemo2 extends JFrame {

    // These variables are used in an inner class so they need
    // to have class scope.
    private FontDialog fontDialog = null;
    private JLabel message = null;

    public DialogDemo2() {
        super("Dialog Demo");
        Container contentPane = getContentPane();

        fontDialog = new FontDialog(this);
        fontDialog.pack();

        // JLabel implements interface SwingConstants which
        // defines constants LEFT, CENTER
        message = new JLabel("Hello World!", JLabel.CENTER);

        // Initialize the font according to the default values in
        // the dialog box.
        updateFont();
        contentPane.add(message);

        // Add menu
        JMenuBar menuBar;
        JMenu menu;
        JMenuItem menuItem;

        menuBar = new JMenuBar();
        setJMenuBar(menuBar);

        // Build the first menu.
        menu = new JMenu("File");
        menu.setMnemonic(KeyEvent.VK_F);
        menuBar.add(menu);

        menuItem = new JMenuItem("Exit",
                                KeyEvent.VK_X);
        menuItem.setMnemonic(KeyEvent.VK_X);
        menuItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                System.exit(0);
            }
        });
        menu.add(menuItem);

        // Build the second menu.
        menu = new JMenu("Format");
        menu.setMnemonic(KeyEvent.VK_F);
        menuBar.add(menu);
    }
}
menuItem = new JMenuItem("Font...",
    KeyEvent.VK_F);
menuItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        int returnVal =
    fontDialog.showFontDialog(DialogDemo2.this);
        if (returnVal == FontDialog.OK) {
            // Reset the font
            updateFont();
        }
    }
});
menu.add(menuItem);

// This menu has two items
menuItem = new JMenuItem("Paragraph...",
    KeyEvent.VK_P);
menuItem.setEnabled(false); // Implement later
menu.add(menuItem);

protected void updateFont() {
    int fontStyle = (fontDialog.isBold()? Font.BOLD: Font.PLAIN);
    Font f = new Font("SansSerif",fontStyle,fontDialog.getFontSize());
    message.setFont(f);
}

public static void main(String[] args) {
    JFrame frame = new DialogDemo2();

    frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });

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

// Format/Font Dialog Box
class FontDialog extends JDialog {
    public static final int OK = 1;
    public static final int CANCEL = 2;

    private JCheckBox boldCheckBox;
    private JComboBox fontSizeComboBox;

    // Save the status of the dialog box controls between invocations
    private boolean isBold = false;
    private String fontSize = "12";
    private int status;

    public FontDialog(JFrame owner) {
        // true = modal (ie application is disabled while this window
        // has focus)
super(owner,"Font",true);

Container contentPane = getContentPane();

addWindowListener(new WindowAdapter() {
    // A close event is sent here AND the default
    // close operation is also done. You can't
    // trap the close operations here to prevent the
    // window from closing.
    public void windowClosing(WindowEvent e) {
      cancelSelection();
    }
});

// Lay out the controls top-to-bottom using a Box Layout
JPanel windowPane = new JPanel();
windowPane.setLayout(new BoxLayout(windowPane, BoxLayout.Y_AXIS));
windowPane.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

boldCheckBox = new JCheckBox("Bold");
boldCheckBox.setSelected(isBold);

String opsString[] = {"8","10","12","14","28"};
fontSizeComboBox = new JComboBox(opsString);
fontSizeComboBox.setSelectedItem(fontSize);

JPanel buttonPane = new JPanel();
buttonPane.setLayout(new BoxLayout(buttonPane, BoxLayout.X_AXIS));
buttonPane.setBorder(BorderFactory.createEmptyBorder(0, 10, 10, 10));

buttonPane.add(Box.createHorizontalGlue());
JButton cancelButton = new JButton("Cancel");
cancelButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      cancelSelection();
    }
});
buttonPane.add(cancelButton);
// Leave some space between buttons
buttonPane.add(Box.createRigidArea(new Dimension(10, 0)));

JButton connectButton = new JButton("OK");
connectButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      approveSelection();
    }
});
buttonPane.add(connectButton);

windowPane.add(boldCheckBox);
windowPane.add(fontSizeComboBox);
windowPane.add(buttonPane);

contentPane.add(windowPane,BorderLayout.CENTER);

protected void cancelSelection() {
// Don't save value of UI elements
status = CANCEL;
setVisible(false);
}

protected void approveSelection() {
    // Save value of UI elements
    fontSize = (String)fontSizeComboBox.getSelectedItem();
    isBold = boldCheckBox.isSelected();

    status = OK;
    setVisible(false);
}

public int showFontDialog(Component parent) {
    // Reset UI elements from instance variables
    boldCheckBox.setSelected(isBold);
    fontSizeComboBox.setSelectedItem(fontSize);

    // Position the dialog box
    setLocationRelativeTo(parent);

    show();
    return status;
}

public boolean isBold() {
    return isBold;
}

public int getFontSize() {
    return Integer.parseInt(fontSize);
}

Extra for Experts

Changing the Window's Icon

The following program demonstrates how to change a window's icon.

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

public class FrameWithNewIcon extends JFrame {
    public FrameWithNewIcon() {
        setTitle("Hello World");
        setSize(300, 200);

        Image image = Toolkit.getDefaultToolkit().getImage("myicon.gif");
    }
```

MediaTracker tracker = new MediaTracker(this);
tracker.addImage(image, 0);

try {
    tracker.waitForID(0);
} catch (Exception e) {
    System.out.println(e);
}

setIconImage(image);

// OR...
// ImageIcon imageIcon = new ImageIcon("myicon.gif");
// setDefaultCloseOperation(imageIcon.getImage());
// ImageIcon will create a media tracker for you

addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});

Container contentPane = getContentPane();
DisplayArea da = new DisplayArea();
contentPane.add(da, BorderLayout.CENTER);

public static void main(String[] args) {
    JFrame frame = new FrameWithNewIcon();
    frame.show();
}

class DisplayArea extends JPanel {

    public void paintComponent(Graphics g) {
        // This will repaint the background
        super.paintComponent(g);

        String helloWorld = "Hello World";
        FontMetrics fm = g.getFontMetrics();
        int stringWidth = fm.stringWidth(helloWorld);
        int stringHeight = fm.getHeight();
        int x = (getWidth() - stringWidth)/2;
        int y = (getHeight() - stringHeight)/2;
        g.drawString(helloWorld, x, y);
    }
}

http://java.sun.com/docs/books/tutorial/uiswing/components/menu.html
http://iglwww.epfl.ch/teaching/software_engineering/documentation/project/swing-crash-course.pdf