Chapter 13 : Creating Windows & Applets

The Java 1.0 *Abstract Window Toolkit* (AWT) produces a GUI that looks equally mediocre on all systems. In addition, it's restrictive.

The situation improved with the Java 1.1 AWT event model.

Java 2 finishes the transformation by essentially replacing everything with the *Java Foundation Classes* (JFC), the GUI portion of which is called "Swing".

These are a rich set of easy-to-use, easy-to-understand JavaBeans that can be dragged and dropped (as well as hand programmed) to create a GUI that you can be satisfied with.

As you learn about Swing you'll discover:

- ❖ Swing is a much better programming model than you've probably seen. JavaBeans is the framework for that library.
- ❖ "GUI builders" (visual programming environments) are a *de rigueur* aspect of a complete Java development environment.
- ❖ The simplicity and well-designed nature of Swing means that even if you do use a GUI builder, the resulting code will still be comprehensible.

For speed, all the components are "lightweight", and Swing is written entirely in Java for portability.

Keyboard navigation is automatic. And scrolling support is effortless.

13.1 The basic applet

Applets are little programs that run inside a Web browser. Because they must be safe, applets are limited in what they can accomplish.

13.1.1 Applet restrictions

An applet is supposed to extend the functionality of a Web page in a browser.

1.*An applet can't touch the local disk*. This means writing *or* reading.

2.*Applets can take longer to display*, since you must download the whole thing every time. ---- JAR files

13.1.2 Applet advantages

1.*There is no installation issue*. An applet has true platform independence.

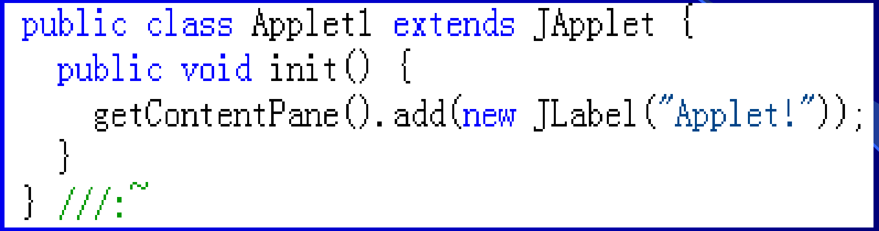
2.*You don't have to worry about bad code causing damage to someone's system*, because of the security built into the core Java language and applet structure.

13.1.3 Application frameworks

A certain category of library is the *application framework*, whose goal is to help you build applications by providing a class or set of classes that produces the basic behavior.

To customize the behavior to your own needs, you should inherit from the application class and override the methods of interest.

Applets are built using an application framework. You inherit from class **JApplet** and override the appropriate methods. ---- init( ), start( ), stop( ), destroy( )



```
public class Applet1 extends JApplet {
  public void init() {
    getContentPane().add(new JLabel("Applet!"));
  }
} ///:~
```

Note : applets are not required to have a **main( )**. That's all wired into the application framework; you put any startup code in **init( )**. (You will often see a leading "**J**" used with Swing components.)

Swing requires you to add all components to the "content pane" of a form, and so you must call **getContentPane( )** as part of the **add( )** process.

13.1.4 Running applets inside a Web browser

Sample : Applet1.html

13.1.5 Using Appletviewer

Appletviewer can pick the **<applet>** tags out of a HTML file and run the applets without displaying the surrounding HTML text.

You can put APPLET tags in the Java source file as comments:

```
// <applet code=MyApplet width=200 height=100>
// </applet>
```

Sample : Applet1b.java

```
appletviewer Applet1b.java
```

13.1.6 Testing applets

As the HTML file is loaded, the browser will discover the applet tag and go hunt for the **.class** file specified by the **code** value.

If your **.class** file isn't in the CLASSPATH then the browser will give an error message.

13.1.6 Running applets from the command line

You can simply add a **main( )** to your applet that builds an instance of the applet inside a **JFrame**. ---- work as both an application and an applet

Sample : Applet1c.java

The applet is explicitly initialized and started in main( ).

13.1.7 A display framework

Sample : Console.java

Java 2 has two ways of causing certain types of windows to close: In JDK 1.2, the solution is to create a new **WindowAdapter** class and implement **windowClosing( )**. However, during the creation of JDK 1.3, the library designers added the **setDefaultCloseOperation( )** to **JFrame** and **JDialog**.

The **run( )** method is overloaded to work with **JApplet**s, **JPanel**s, and **JFrame**s. Note that only if it's a **JApplet** are **init( )** and **start( )** called.

Sample : Applet1d.java

13.1.8 Making a button

Making a button is quite simple: you just call the **JButton** constructor with the label you want on the button.

You don’t need to explicitly paint a button or any other kind of control; you simply place them on the form and let them automatically paint themselves.

Sample : Button1.java

13.1.9 Capturing an event

The basis of event-driven programming, which comprises a lot of what a GUI is about, is tying events to code that responds to those events.

Each Swing component can report all the events that might happen to it, and it can report each kind of event individually.

At first, we will just focus on the main event of interest for the components. In the case of a **JButton**, this “event of interest” is that the button is pressed.

addActionListener(ActionListener l)

$\downarrow$

actionPerformed(ActionEvent e)

Sample : Button2.java $\longrightarrow$ Sample : Button2b.java

It is often more convenient to code the **ActionListener** as an anonymous inner class.

The argument to **actionPerformed( )** is of type **ActionEvent**, which contains all the information about the event and where it came from.

The simplest way to create a **JTextField** is just to tell the constructor how wide you want that field to be.

13.1.10 Text areas

A **JTextArea** is like a **JTextField** except that it can have multiple lines and has more functionality. A particularly useful method is **append( )**.

Sample : TextArea.java

13.2 Controlling layout

The way that you place components on a form in Java is probably different from any other GUI system you’ve used.

1. it's all code; there are no "resources" that control placement of components.

2. The way components are placed on a form is controlled not by absolute positioning but by a "layout manager" based on the order that you **add( )** them.

The size, shape, and placement of components will be remarkably different from one layout manager to another. In addition, the layout managers adapt to the dimensions of your applet or application window.

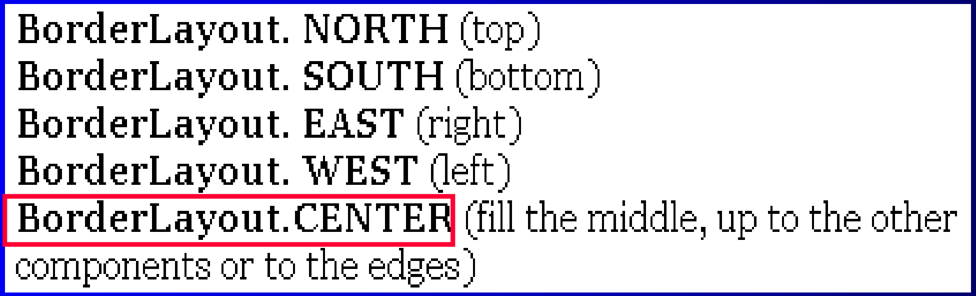
- ✓ **JApplet**, **JFrame**, **JWindow** and **JDialog** can all produce a **Container** with **getContentPane( )** that can contain and display **Components**. In **Container,** the method **setLayout( )** allows you to choose a different layout manager.
- ✓ Other classes (e.g. **JPanel**) contain and display components directly and so you also set the layout manager directly.

13.2.1 BorderLayout

The applet uses a default layout scheme: the **BorderLayout**. Without any other instruction, this takes whatever you **add( )** to it and places it in the center, stretching the object all the way out to the edges.

This layout manager has the concept of four border regions and a center area.

default ←



BorderLayout. NORTH (top)
BorderLayout. SOUTH (bottom)
BorderLayout. EAST (right)
BorderLayout. WEST (left)
BorderLayout.CENTER (fill the middle, up to the other components or to the edges)

Sample : BorderLayout1.java

13.2.2 FlowLayout

This simply “flows” the components onto the form, from left to right until the top space is full, then moves down a row and continues flowing.

With **FlowLayout** the components take on their “natural” size.

Sample : FlowLayout1.java

13.2.3 GridLayout

A **GridLayout** allows you to build a table of components, and as you add them they are placed left-to-right and top-to-bottom in the grid.

The components are laid out in equal proportions.

Sample : GridLayout1.java

13.2.4 GridBagLayout

The **GridBagLayout** provides you with tremendous control in deciding exactly how the regions of your window will lay themselves out.

It is intended primarily for automatic code generation by a GUI builder.

13.2.5 Absolute positioning

- Set a **null** layout manager for your **Container**: **setLayout(null)**.
- Call **setBounds( )** or **reshape( )** for each component.

However, this is usually not the best way to generate code.

13.2.6 BoxLayout

BoxLayout gives you many of the benefits of **GridBagLayout** without the complexity.

BoxLayout allows you to control the placement of components either vertically or horizontally.

There's a special container called **Box** that uses **BoxLayout** as its native manager.

Sample : Box1.java

Struts add space between components, measured in pixels.

Sample : Box2.java

Glue is the opposite: it separates components by as much as possible. Thus it’s more of a “spring” than “glue”.

Sample : Box3.java

13.3 The Swing event model

In the Swing event model a component can initiate an event. Each type of event is represented by a distinct class. When an event is fired, it is received by one or more “listeners”, which act on that event.

Each event listener is an object of a class that implements a particular type of listener **interface**.

You should create a listener object and register it with the component ---- add$\boxed{\text{XXX}}$Listener( )

$\text{XXX} \longrightarrow$ Event type

All of the event logic will go inside a listener class. ---- using an inner class

13.3.1 Event and listener types

All Swing components include **addXXXListener( )** and **removeXXXListener( )** methods so that the appropriate types of listeners can be added and removed from each component.

See the table on Page 723

Each type of component supports only certain types of events.

Sample : ShowAddListeners.java

The steps for dealing with events :

❖ Take the name of the event class and substitute "Listener" to "Event" --- the listener interface

❖ Implement the interface above and write out the methods for the events you want to capture.

❖ Create an object of the listener class in the previous step. Register it with your component.

See the table on Page 728
(some listener interfaces)

Using listener adapters for simplicity :

Some (but not all) of the listener interfaces that have more than one method are provided with *adapters*. Each adapter provides default empty methods for each of the interface methods.

All you need to do is to inherit from the adapter and override only the methods you need to change.

Sample : TrackEvent.java

13.4 Introduction to Swing components

13.4.1 Buttons

All buttons, check boxes, radio buttons, and even menu items are inherited from **AbstractButton**.

Sample : Buttons.java

Button groups :

If you want radio buttons to behave in an "exclusive or" fashion, you must add them to a "button group". Any AbstractButton can be added to a **ButtonGroup**.

Sample : ButtonGroups.java

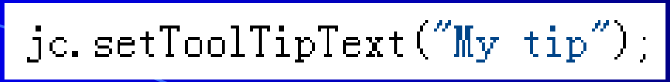
13.4.2 Icons

You can use an **Icon** inside a **JLabel** or anything that inherits from **AbstractButton**.

Sample : Faces.java

13.4.3 Tool tips

Almost all of the classes that you'll be using to create your user interfaces are derived from **JComponent**, which contains a method called **setToolTipText(String)**.



```
jc.setToolTipText("My tip");
```

13.4.4 Text fields

Sample : TextFields.java

13.4.5 Borders

JComponent contains a method called **setBorder( )**, which allows you to place various interesting borders on any visible component.

Sample : Borders.java

13.4.6 JScrollPanes

Sample : JScrollPanes.java

13.4.7 Check boxes

You'll normally create a **JCheckBox** using a constructor that takes the label as an argument. You can get and set the state, and also get and set the label if you want to read or change it after the **JCheckBox** has been created.

Whenever a **JCheckBox** is set or cleared, an event occurs.

Sample : CheckBoxes.java

13.4.8 Radio buttons

One of the buttons can optionally have its starting state set to true. If you try to set more than one radio button to **true** then only the final one set will be true.

Sample : RadioButtons.java

13.4.9 Combo boxes

Windows combo box lets you select from a list *or* type in your own selection. With a **JComboBox** box you can choose only one element from the list.

Sample : ComboBoxes.java

13.4.10 List boxes

A **JList** occupies some fixed number of lines on a screen and doesn't change. If you want to see the selected items in a list, you can simply call getSelectedValues( ). A **JList** allows multiple selection.

Sample : List.java

13.4.11 Message boxes

Windowing environments commonly contain a standard set of message boxes that allow you to quickly post information to the user or to obtain information from the user. ----JOptionPane

Sample : MessageBoxes.java

13.4.12 Menus

Each component capable of holding a menu, including **JApplet**, **JFrame**, **JDialog**, and their descendants, has a **setJMenuBar( )** method that accepts a **JMenuBar**.

You can add **JMenus** to the **JMenuBar**, and **JMenuItems** to the **JMenus**. Each **JMenuItem** can have an **ActionListener** attached to it.

Unlike a system that uses resources, with Java and Swing you must hand assemble all the menus in source code.

Sample : SimpleMenus.java

There are three types inherited from **JMenuItem** :

- JMenu
- JCheckBoxMenuItem
- JRadioButtonMenuItem

Swing supports mnemonics, or “keyboard shortcuts”, so you can select anything derived from **AbstractButton** using the keyboard instead of the mouse.

Sample : Menus.java

13.4.13 Pop-up Menus

Sample : Popup.java

13.4.14 Drawing

If you want a straightforward drawing surface you will typically inherit from a **JPanel**. The only method you need to override is **paintComponent( )**, which is called whenever that component must be repainted.

Use the **Graphics** methods that you can find in the documentation for **java.awt.Graphics** .

Sample : SineWave.java

13.4.15 Dialog Boxes

Dialog boxes are heavily used in windowed programming environments, but less frequently used in applets.

JDialog ---- One significant difference when **windowClosing( )** is called is that you don’t want to shut down the application. Instead, you release the resources by calling **dispose( )**.

Sample : Dialogs.java

13.4.16 Trees

The API for trees is vast. But the “default” tree components can generally do what you need.

The **JTree** is controlled through its *model*. When you make a change to the model, the model generates an event that causes the **JTree** to perform any necessary updates.

Sample : Trees.java

13.4.17 Tables

The **JTable** controls how the data is displayed, but the **TableModel** controls the data itself. So to create a **JTable** you'll typically create a **TableModel** first.

Sample : Table.java

13.4.18 Selecting Look & Feel

"Pluggable Look & Feel" allows your program to emulate the look and feel of various operating environments.

- ✓ select the "cross platform" look and feel ("metal").
- ✓ select the look and feel for the current system.

If you want to use the cross-platform ("metal") look and feel, you don't have to do anything.

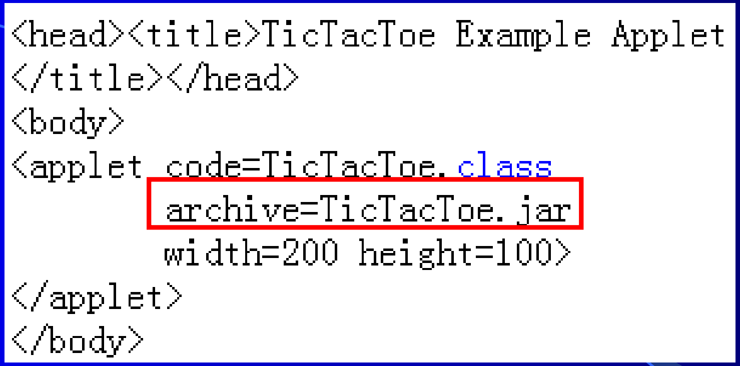
If you want to use the current operating environment’s look and feel, you must insert the following code, typically at the beginning of your **main( )** before any components are added.

```
try {
  UIManager.setLookAndFeel(UIManager.
    getSystemLookAndFeelClassName());
} catch(Exception e) {}
```

Sample : LookAndFeel.java

13.5 Packaging an applet into a JAR file

```
jar cf TicTacToe.jar *.class
```



```
<head><title>TicTacToe Example Applet
</title></head>
<body>
<applet code=TicTacToe.class
        archive=TicTacToe.jar
        width=200 height=100>
</applet>
</body>
```

Sample : TicTacToe

13.6 Binding events dynamically

Sample : DynamicEvents.java

Usually, components handle events as *multicast*. ---- You can register many listeners for a single event.

Note : Event listeners are not guaranteed to be called in the order they are added.

13.7 Visual programming and Beans

Part of the process of visual programming involves dragging a component from a palette and dropping it onto your form.

The application builder tool uses reflection to dynamically interrogate the component and find out which properties and events the component supports.

13.7.1 What is a Bean?

A component is really just a block of code, typically embodied in a class.

Java has brought the creation of visual components to its most advanced state with JavaBeans, because a Bean is just a class.

You don’t have to write any extra code or use special language extensions in order to make something a Bean.

It is the method name that tells the application builder tool whether this is a property, an event, or just an ordinary method.

- For a property named **xxx**, you typically create two methods: **getXxx( )** and **setXxx( )**.
- For a **boolean** property, you can use the “get” and “set” approach above, but you can also use “is” instead of “get”.
- Ordinary methods of the Bean don’t conform to the above naming convention, but they’re **public**.
- For events, you use the Swing “listener” approach.

Sample : Frog.java

The property name does not force you to use any particular identifier for internal variables (or to even *have* any internal variables for that property).

13.7.2 Extracting BeanInfo with the Introspector

The application builder tool must be able to create the Bean and then, without access to the Bean's source code, extract all the necessary information to create the property sheet and event handlers.

Introspector.getBeanInfo( ) : You can pass a **Class** reference to this method and it fully interrogates that class and returns a **BeanInfo** object.

Sample : BeanDumper.java

Sample : BangBean.java

BangBean implements the **Serializable** interface. This means that the application builder tool can save all the information for the **BangBean** using serialization after the program designer has adjusted the values of the properties.

All the fields are **private**, which is what you'll usually do with a Bean.

13.7.3 Packaging a Bean

Before you can bring a Bean into a Bean-enabled visual builder tool, it must be put into the standard Bean container, which is a JAR file that includes all the Bean classes as well as a "manifest" file.

A manifest file is simply a text file that follows a particular form.

```
//:! :BangBean.mf
Manifest-Version: 1.0

Name: bangbean/BangBean.class
Java-Bean: True
///:~
```

The name in the manifest file must include the package information. In addition, you must place the manifest file in the directory *above* the root of your package path.

```
jar cfm BangBean.jar BangBean.mf bangbean
```

You can also add other Beans to the JAR file simply by adding their information to your manifest.

You'll probably want to put each Bean in its own subdirectory.