Applets and GUIs

COMS W1007
Introduction to Computer Science

Christopher Conway
19 June 2003

GUIs in Java

There are at least three separate methods for doing GUIs in Java:

- Abstract Window Toolkit (AWT) 1.0 (1995)
- AWT 1.1 (1997)
- Swing (1999)

Many people still use AWT 1.0 for maximum compatability. We will use Swing, because that's what's in the book.

GUIs in Java: 2

The long evolution of Java's GUI tools means we have to dig through a lot of packages to find what we need.

- **`java.awt`** - Swing is built on top of AWT. All of the top-level classes and many of the utility classes come from the AWT package.
- **`java.awt.event`** - the Java event model (more later) has been consistent since AWT 1.1.
- **`javax.swing`** - all of the Swing-specific classes are here, including all the actual GUI components.

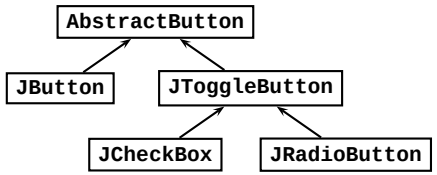
GUI Components: A Taxonomy

The Java GUI toolkits provide the full assortment of GUI components that you find on every platform. Types of components include:

- Buttons, Check Boxes and Radio Buttons
- Combo/List Boxes
- Text Areas and Fields
- Scroll Bars

Buttons

Buttons are simple components with one basic capability: you can click on them. The Swing button classes are:



Toggle Buttons

A regular button has no state. It is just a portion of the screen you can click on.

A toggle button has a "checked" and "unchecked" state.

- A *check box* is a button you can check and uncheck.
- A *radio button* is part of a group of buttons, only one of which can be checked at the same time.

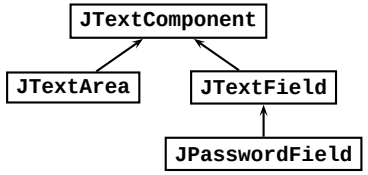
Combo Boxes

A *combo box* or *list box* is a drop-down list of items, only one of which may be selected.

The Swing combo box class is **`JComboBox`**.

Text Components

Text components are areas for entering and displaying text. The Swing text component classes are:



Text Components: 2

- A *text field* allows the input of a single line of text.
- A *text area* allows the input of multiple lines of text.
- A *password field* is a text field that masks the input (usually with '*').

Text components can be editable (the user can type in it) or non-editable (only the programmer can put text in it).

Scroll Bars

Scroll bars allow a component to contain more data than can be displayed on screen at once. The Swing class that handles scrolling is **`JScrollPane`**.

`JScrollPane` is a container: it is a component into which you place other components. The components inside the scroll pane handle their own behavior and data, and the scroll pane provides a *viewport* onto the components, which can be controlled using the scroll bars.

Panels

Another type of container is a panel. A panel doesn't have any behavior of its own; it just provides a place to put other components. We often use panels when we layout control. The Swing panel class is **`JPanel`**.

The parent class of all GUI containers is **`java.awt.Container`**. **`Container`** has a method **`add(Component c)`** that adds a component to the container.

Layout Managers

The placement of components inside a container is controlled by the layout manager. The **`java.awt`** package has an interface **`LayoutManager`** which is implemented by:

- **`FlowLayout`** - components are placed from left to right and wrap when there's no more room, like text in a paragraph.
- **`GridLayout`** - components are placed in a rigid $m \times n$ grid.

Layout Managers, cont'd

- **`BorderLayout`** - each component is specified as at the top, bottom, left, right or center of the container.
- **`GridBagLayout`** - the programmer specifies in great detail exactly where every component should go.

`GridBagLayout` is by far the most flexible, and the most complicated. If you want to know more about layout managers, read the book.

`Container` has a method **`add(Component c, Object constraints)`** that specifies constraints to the layout manager. The type of **`constraints`** is defined by the implementation of **`LayoutManager`**.

Applets

An *applet* is a Java program that can run inside a web browser. To write a Swing applet, you create a class that extends **`javax.swing.JApplet`**. If your applet doesn't use Swing (unlikely), you can extend **`java.applet.Applet`** instead. **`Applet`** is the parent of **`JApplet`**.

When you run a Java program from the command line, the JVM looks for a **`main`** method to execute. When you run an applet in a web browser, the JVM looks for methods inherited from **`Applet`**: **`init`**, **`start`**, **`stop`** and **`destroy`**.

Applet Methods

- **`init`** - this method is called when the applet is loaded (i.e., brought into memory).
- **`start`** - this method is called when the applet starts. If a user leaves the applet's web page and comes back, the applet will load once and start twice.
- **`stop`** - this method is called when the applet stops (e.g., when the user leaves the applet's web page).
- **`destroy`** - this method is called when the applet is unloaded.

Applet Methods: 2

If the applet were a videotape, these events would correspond to:

- **`init`** - putting the tape in the VCR
- **`start`** - pressing play
- **`stop`** - pressing stop
- **`destroy`** - ejecting the tape

Applet Methods: 3

We don't have to override all of these methods; in fact, we usually don't.

`init` plays the same role as a constructor. If the applet has GUI components (and it usually does), we can create them and lay them out in **`init`**.

`start` and **`stop`** are useful if the applet has a real-time component (e.g., an animation).

We almost never override **`destroy`**.

The Applet Container

Every applet has an associatede GUI container.

- **`JApplet`** has a method **`getContentPane()`** that returns a **`Container`**. You can add components to this container.

```
Container c = getContentPane() ;
c.add( new JButton("Click Here!") ) ;
```

- **`Applet`** is *itself* a **`Container`**—it extends **`java.awt.Panel`**. Components are added to the applet itself.

```
this.add( new JButton("OK") ) ;
```

The <applet> Tag

An applet is embedded in a web page using the HTML **<applet>** tag. The tag has attributes to define the applet class and the dimensions of the applet area on the page:

```
<applet code="MyApplet.class" width=240
    height=120></applet>
```

Text that appears between start and end tags will only be displayed if the web browser doesn't support applets.

```
<applet code="MyApplet.class" width=240
    height=120>
  You browser is ignoring the &lt;applet&gt;
  tag. Consider upgrading, or just go away.
</applet>
```

Applet Parameters

You can specify "command-line" arguments to an applet by using the **<param>** tag inside the **<applet>** tag. You can read the parameters in the applet using the **getParameter** method of the **Applet** class.

```
<applet code="MyApplet.class" width=240
    height=120>
  <param name="file" value="input.txt" />
</applet>
```

```
String in = getParameter("file") ;
  /* in="input.txt" */
```

The AWT Event Model

User input in a GUI environment is handled *asynchronously*. We call a user action an *event*.

Objects (*event listeners*) register their interest in a certain type of event (clicking, pressing a key, moving the mouse) with the object that may generate the event (buttons, text fields, containers).

When an event occurs, the generating object invokes a method (an *event handler*) on each registered listener.

The EventListener Interface

Event listeners implement a subinterface of **java.util.EventListener**. Some common listener interfaces are:

- **MouseListener** - reacts to mouse button events (button down, button up and click (down+up) are separate events).
- **MouseMotionListener** - reacts to mouse movement and dragging (motion with a button pressed).
- **KeyListener** - reacts to keyboard events (key down, key up and key typed (down+up)).
- **TextListener** - reacts to a change in the value of a text component.
- **ItemListener** - reacts in a change to the state of a component (e.g., the selection in a combo box changes).
- **ActionListener** - reacts to an "action" on a component (e.g., clicking a button). What an action is depends on the component.
- **FocusListener** - reacts to a change in focus (i.e., which component is currently active).
- **WindowListener** - reacts to a change in the state of a window (e.g., an attempt to close it).

Adapter Classes

Sometimes we only want to deal with one particular event, but the listener interface handles many related events. **java.awt.event** contains adapter classes that implement the listener interfaces with do-nothing methods. We can extend an adapter class and override just the methods that we are interested in.

Some adapter classes are:

- **MouseAdapter**
- **MouseMotionAdapter**
- **KeyAdapter**
- **FocusAdapter**
- **WindowAdapter**

```
public class Word extends JApplet
        implements ActionListener {
  JTextField textfield ;

  public void init() {
    textfield = new JTextField() ;
    JButton button = new JButton("Add A Word") ;
    button.addActionListener( this ) ;
    ...
  }

  public void actionPerformed(ActionEvent e) {
    if( e.getID()==ActionEvent.ACTION_PERFORMED )
      textfield.setText( textfield.getText()
              + "A Word") ;
  }
}
```

Inner Classes

If your application has a lot of GUI components, the number of listener interfaces implemented becomes unwieldy and annoying. At the same time, defining a separate class for each listener is inconvenient.

An *inner class* is a class declared inside another class. If an inner class is a member of the *outer class*, it can access the other members of the class.

A *local inner class* is a class declared inside a code block, e.g., a method body. A local inner class can access variables inside the scope of the block, as long as they are declared **final**.

```
public class InnerWord extends JApplet {
  JTextField textfield ;

  class ActionHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if( e.getID()==ActionEvent.ACTION_PERFORMED )
        textfield.setText( textfield.getText()
                + "A Word") ;
    }
  }

  public void init() {
    textfield = new JTextField() ;
    JButton button = new JButton("Add A Word") ;
    button.addActionListener( new ActionHandler() ) ;
    ...
  }
}
```

```
public class LocalInnerWord extends JApplet {
  public void init() {
    final JTextField textfield = new JTextField() ;
    JButton button = new JButton("Add A Word") ;

    class ActionHandler implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        if( e.getID()==ActionEvent.ACTION_PERFORMED )
          textfield.setText( textfield.getText()
                  + "A Word") ;
      }
    }
    button.addActionListener( new ActionHandler() ) ;
   ...
  }
}
```

Anonymous Inner Classes

If an inner class is only used once, giving an it a name is a waste of time. An anonymous inner class can be created using the syntax:

```
new AbstractClassName() {
  /* abstract method definitions */
}
```

or

```
new InterfaceName() {
  /* interface method definitions */
}
```

```
public class AnonymousWord extends JApplet {
  public void init() {
    final JTextField textfield = new JTextField() ;
    JButton button = new JButton("Add A Word") ;

    button.addActionListener( new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if( e.getID()==ActionEvent.ACTION_PERFORMED )
          textfield.setText( textfield.getText()
                  + "A Word") ;
      }
    } ) ;
    ...
  }
}
```

The `JComboBox` component

The **`JComboBox`** component operates on a list of objects. Each item is displayed in the list using the object's **`toString`** method.

The method **`getSelectedItem`** returns the **`Object`** in the list that is selected. If you know the type of the item (and you should), you can case the **`Object`** to its proper type and use it.

An **`ItemListener`** on a **`JComboBox`** will be called when the selection changes.