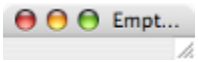
Programming via Java Swing basics

Most programs that people use from day to day involve a graphical user interface, a term which is abbreviated as *GUI* and often pronounced *gooey*. We're going to look at how to build a GUI using a Java library called Swing.

Swing isn't the only Java library for GUIs: There is another prominent one included with Java called AWT (for Abstract Window Toolkit), and some developers instead use SWT, a GUI library that doesn't come with Java. In fact, Swing is built using AWT, and we'll see that Swing programs often manipulate some AWT classes.

Historically, AWT came first. It was an attempt to provide a set of classes that interfaced with the graphical elements provided by each operating system. Thus, if we create a button using AWT, it corresponds to a MacOS X button when the program is run in MacOS X, a Linux button when run under Linux, and a Windows button when run under Windows. This leads to difficulties due to slight differences in behavior under different operating systems. Rather than continue to deal with the resulting headaches, Java's developers created Swing, which implements all the graphical elements entirely in Java.

24.1. The simplest GUI



The simplest Swing program, in Figure 24.1, simply displays an empty window. It uses just one Swing class, `JFrame`, which represents a window on the screen.

Figure 24.1: The `EmptyWindow` program.

```
1  import javax.swing.JFrame;
2
3  public class EmptyWindow {
4      public static void main(String[] args) {
5          JFrame frame = new JFrame("Empty Window");
6          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7          frame.pack();
8          frame.setVisible(true);
9      }
10 }
```

A `JFrame` has just a few methods worth talking about for now.

```
JFrame(String title)
```

(Constructor) Creates an object representing a window with `title` in its title bar. Note that the window doesn't itself appear on the screen yet; that is done using its `setVisible` method.

```
void setDefaultCloseOperation(int value)
```

Configures how this window behaves when the user clicks on the close-window button in the title bar. For simple programs you will want this to be`JFrame.EXIT_ON_CLOSE`. The default behavior is usually `JFrame.HIDE_ON_CLOSE`, where the window disappears from the screen but the base program continues to execute.

(More complex programs often want specialized behavior, such as showing a confirmation dialog. To do this, they use `JFrame.DO_NOTHING_ON_CLOSE` with this method and also add a `WindowListener` to the `JFrame` with a `windowClosing` method. We'll get to talking about listeners later.)

```
void pack()
```

Resizes this window to accommodate the components within it.

```
void setVisible(boolean value)
```

Configures whether this window is visible on the screen.

```
Container getContentPane()
```

Returns an object into which you can place GUI elements that should appear in the window.

24.2. Inserting components

A `Container` is an object used for holding elements which might appear in a graphical user interface. We specify which elements go into it using either of its two methods named `add`.

```
void add(Component what)
```

Inserts `what` into this container.

```
void add(Component what, Object info)
```

Inserts `what` into this container, using `info` as information about where the object should be placed.

The first parameter for each method is a `Component` representing an element that appears in a window. You'll never write **`new`** `Component()` to create a `Component` instance. Instead, you'll create `Component`'s subclasses. Right now, we'll look at two subclasses: `JButton` and `JTextField`.

`JButton(String label)`

(Constructor) Creates a button holding the string `label`. A **button** is an area the user can click for an action to occur; you see them most often at the bottom of dialog boxes containing labels such as OK or Cancel. We'll see how to get the buttons to do something in Section 24.4.

`JTextField(int columns)`

(Constructor) Creates a text field, sized to contain `columns` characters. A **text field** is an area where the user can type a line of text.

`String getText()`

Returns the current text inside this field.

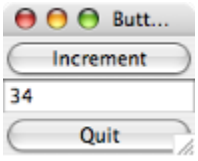
`void setText(String text)`

Changes this field's current text to `text`.

The second `add` method takes a second parameter. This parameter is important: Without it, each component is placed on top of each other in the window's center, and the window isn't very useful. The possible values for this second parameter include five values indicating to place the added component on one of the window's four edges or in the window's center. (Section 24.5.2 will describe more ways to place components within a window.)

`BorderLayout.NORTH`

`BorderLayout.WEST` `BorderLayout.CENTER` `BorderLayout.EAST`



`BorderLayout.SOUTH`

Using `JButton` and `JTextField`, we can build a program that creates a window containing a text field sandwiched by two buttons labeled Increment andQuit. field. We're going to work

on making the buttons *do* something soon. For the moment, all the program does is display the components.

Figure 24.2: The `ButtonExample` program.

```
1  import java.awt.BorderLayout;  import java.awt.Container;
2  import javax.swing.JButton;    import javax.swing.JFrame;
3  import javax.swing.JTextField;
4
5  public class ButtonExample {
6      public static void main(String[] args) {
7          JButton incrButton = new JButton("Increment");
8          JButton quitButton = new JButton("Quit");
9          JTextField numberField = new JTextField();
10
11         JFrame frame = new JFrame("Button Example");
12         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13         Container contents = frame.getContentPane();
14         contents.add(incrButton, BorderLayout.NORTH);
15         contents.add(numberField, BorderLayout.CENTER);
16         contents.add(quitButton, BorderLayout.SOUTH);
17         frame.pack();
18         frame.setVisible(true);
19     }
20 }
```

This program has two differences from the `EmptyWindow` program from before. In lines 7 to 9 we create objects corresponding to the two buttons and text field. This simply creates objects, but they are not yet part of the window. Lines 13 to 16 adds the components into the window's content pane. Notice that as we add each component, we include a second parameter indicating where to place it within the window.

24.3. Interfaces

Now we can work on allowing the user to interact with the window. In order to do this, we need to discuss interfaces. An **interface** is basically just a set of methods. Any class that hopes to **implement** the interface must implement all of the methods in the interface's set.

Despite the name, interfaces are not specific to graphical user interfaces! They're a general Java concept, that we just happen to be discussing in the context of graphical user interfaces.

Defining an interface looks a lot like defining a class, except that we can only list instance methods, and none of the instance methods have any bodies. In place of each instance method's body, we just place a semicolon.

For example, we might have defined the following interface for classes that have a position.

```
public interface Locatable {
    public double getX();
    public double getY();
}
```

With the interface defined, we will now need classes that claim to implement the interface. For this, we need to say so up front using an **`implements`** clause.

```
public class Ball implements Locatable { //...
public class Paddle extends Block implements Locatable { //...
```

Any class that has such an **`implements`** clause has to define all of the methods defined in the interface. Otherwise, there's a compiler error. Thus, both `Ball` and `Paddle`must have `getX` and `getY` methods; since the interface says these methods take no parameters and return a **`double`**, they need to be defined this way in `Ball` and `Paddle`too.

What makes interfaces useful is that they are types, too. The following would be completely legitimate.

```
Locatable loc = new Ball();
System.out.println(loc.getX() + ", " + loc.getY());
loc = new Paddle();
System.out.println(loc.getX() + ", " + loc.getY());
```

You cannot create an instance of an interface: **`new`** `Locatable()` would be illegal.

In some sense, with interfaces a class have two superclasses. Only one of these can be a true Java class, but the type conversion behavior makes it feel as if both the actual parent class and the implemented interface are superclasses.

24.4. Event listeners

So how can we have our button do something? We're going to have to use an interface — namely, the `ActionListener` interface in the `java.awt.event` package, which includes the following method.

```
public void actionPerformed(ActionEvent evt)
```

> Called when an action is performed on a GUI component. In the case of a button, the action occurs when the user clicks on it.

The `ActionEvent` parameter specifies information about the event that has occurred. For the moment, we're just going to worry about its `getSource` method, which returns the `Component` on which the action event was performed.

In order to define what to do for the button, we're going to need a class that implements this interface. Inside that class's `actionPerformed` method, we'll have some code that say what to do.

The other step in setting up the program to handle events is to register the ActionListener object with the component. In our case, we'll want the `addActionListener`method in the JButton class. It takes takes an ActionListener as an argument and sets up the button so that, when clicked, it calls the `actionPerformed` method in that ActionListener.

The program of Figure 24.3 illustrates how a program can react to button clicks by implementing `ActionListener`.

Figure 24.3: The `ListeningExample` program.

```
1  import java.awt.BorderLayout;      import java.awt.Container;
2  import java.awt.event.ActionEvent; import java.awt.event.ActionListener;
3  import javax.swing.JButton;        import javax.swing.JFrame;
4  import javax.swing.JTextField;
5
6  public class ListeningExample implements ActionListener {
7      private JButton incr;
8      private JButton quit;
9      private JTextField field;
10
11     private ListeningExample(JButton i, JButton q, JTextField f) {
12         incr = i; quit = q; field = f;
13     }
14
```

```
15      public void actionPerformed(ActionEvent e) {
16          if(e.getSource() == incr) {
17              int i = Integer.parseInt(field.getText());
18              field.setText("" + (i + 1));
19          } else if(e.getSource() == quit) {
20              System.exit(0);
21          }
22      }
23
24      public static void main(String[] args) {
25          JButton incrButton = new JButton("Increment");
26          JButton quitButton = new JButton("Quit");
27          JTextField numberField = new JTextField();
28
29          ListeningExample listener = new ListeningExample(incrButton,
30              quitButton, numberField);
31          incrButton.addActionListener(listener);
32          quitButton.addActionListener(listener);
33
34          JFrame frame = new JFrame("Listening Example");
35          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
36          Container contents = frame.getContentPane();
37          contents.add(incrButton, BorderLayout.NORTH);
38          contents.add(numberField, BorderLayout.CENTER);
39          contents.add(quitButton, BorderLayout.SOUTH);
40          frame.pack();
41          frame.setVisible(true);
42      }
43  }
```

The important changes here are that we've added two lines in the `main` method using the `addActionListener` method to register the `ListeningExample` object to listen to actions being performed on its two buttons. That is, when somebody clicks on either button, it will call the `actionPerformed` method on the constructed `ListeningExample`object.

The other important change is the addition of the `actionPerformed` method. In our case, this method first sees of the components triggered the action — that is, whether the Increment or Quit button was clicked. It performs an action based on this.

24.5. Composing GUIs

We don't yet have the ability to build complex windows: We can create components, and we can place them on the north, south, east, or west edge of the window; or we can place it in the window's center. That limits us to five components, and they're not necessarily placed how we want them.

24.5.1. The `JPanel` class

Building more complex interfaces requires using the `JPanel` class. A `JPanel` object is an empty graphical component, but it is a subclass of `Container`, so you can add other components into it.

There are four JPanel methods that are particularly important.

`JPanel()`

> (Constructor) Creates an empty JPanel.

`void add(Component what)`

> Inserts `what` into this panel, using the default placement.

`void add(Component what, Object info)`

> Inserts `what` into this panel, using `info` as information about where the object should be placed.

`void setLayout(LayoutManager manager)`

> Configures this container to use the indicated technique for laying out its components. We'll see how this works soon.

24.5.2. Layouts

There are several categories of classes involved in creating GUIs using Swing, and we've covered all but the last.

- the `JFrame` class
- component classes (`JButton`, `JTextField`, `JPanel`)
- interfaces for listeners (`ActionListener`)
- event classes (`ActionEvent`)
- container classes (`Container`, `JPanel`)
- classes for managing layouts (`FlowLayout`, `BorderLayout`, `GridLayout`)

This last category consists of classes implementing Java's `LayoutManager` interface. The layout classes tell a container class how it should arrange the components it contains. We can use the `setLayout` method to configure a container, such as `Container` or `JPanel`, to use a different approach for arranging its components.

BorderLayout

We've actually already seen the `BorderLayout` class in passing: It's the default layout for a `JFrame`'s container. Creating a `BorderLayout` object is simple:

```
BorderLayout()
```

> (Constructor) Creates a BorderLayout object.

When you add something to a container that's using the BorderLayout, you will generally use the `add` method that takes an `info` parameter, and you will pass something like `BorderLayout.NORTH` saying on which border to place the component (or `BorderLayout.CENTER` to place the component in the middle).

FlowLayout

The FlowLayout class is even simpler: It places the components in left-to-right order. Each component gets sized to its preferred size. When the components fill the row, they start being placed in the next row.

```
FlowLayout()
```

> (Constructor) Creates a FlowLayout object.

With a container using FlowLayout, you'll generally use the `add` that *doesn't* take an `info` parameter.

FlowLayout is the default layout for a JPanel.

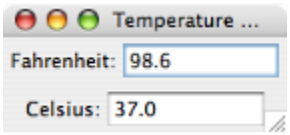
GridLayout

In the GridLayout class, components are placed left-right, top-down in a strict grid. Each component is resized to fill its grid space.

```
GridLayout(int rows, int columns)
```

> (Constructor) Creates a GridLayout object, represent the layout strategy with `rows` rows and `columns` columns.

24.5.3. Case study: Temperature conversion



As an example illustrating the use of `JPanel`, let's work on a program that creates a window as at right. In this program, the user types a temperature into either blank and presses Enter; the converted temperature then appears in the other blank.

To accomplish this, the program creates a `JFrame` whose content pane contains two `JPanel` objects, one on its north edge, one on its south edge. Each `JPanel` uses its default `FlowLayout` manager and contains a `JLabel` and a `JTextField`.

Figure 24.4: The `TempConvert` program.

```
1  import java.awt.BorderLayout;      import java.awt.Container;
2  import java.awt.event.ActionEvent; import java.awt.event.ActionListener;
3  import javax.swing.JButton;        import javax.swing.JFrame;
4  import javax.swing.JLabel;         import javax.swing.JPanel;
5  import javax.swing.JTextField;
6
7  public class TempConvert implements ActionListener {
8      private JTextField fahr;
9      private JTextField cels;
10
11     private TempConvert(JTextField fahr, JTextField cels) {
12         this.fahr = fahr; this.cels = cels;
13     }
14
15     public void actionPerformed(ActionEvent e) {
16         if(e.getSource() == fahr) {
17             double temp = Double.parseDouble(fahr.getText());
18             double val = (temp - 32) / 1.8;
19             cels.setText("" + Math.rint(val * 100) / 100);
20         } else if(e.getSource() == cels) {
21             double temp = Double.parseDouble(cels.getText());
22             double val = 1.8 * temp + 32;
23             fahr.setText("" + Math.rint(val * 100) / 100);
24         }
25     }
26
27     public static void main(String[] args) {
28         JTextField fahrField = new JTextField(8);
```

```
29          JTextField celsField = new JTextField(8);
30
31          JPanel fahrPanel = new JPanel();
32          fahrPanel.add(new JLabel("Fahrenheit:"));
33          fahrPanel.add(fahrField);
34
35          JPanel celsPanel = new JPanel();
36          celsPanel.add(new JLabel("Celsius:"));
37          celsPanel.add(celsField);
38
39          TempConvert listener = new TempConvert(fahrField, celsField);
40          fahrField.addActionListener(listener);
41          celsField.addActionListener(listener);
42
43          JFrame frame = new JFrame("Temperature Conversion");
44          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
45          Container contents = frame.getContentPane();
46          contents.add(fahrPanel, BorderLayout.NORTH);
47          contents.add(celsPanel, BorderLayout.SOUTH);
48          frame.pack();
49          frame.setVisible(true);
50      }
51  }
```

Lines 40 and 41: The `JTextField` action listener

Notice how we're defining an `ActionListener` for a text field in lines 40 and 41. If you add an `ActionListener` to a text field, its `actionPerformed` method occurs whenever the user presses the Enter key into the text field.

Lines 32 and 36: The `JLabel` class

Another new thing in this program is its use of `JLabel`, a component that simply displays an uneditable string to appear in the window. We use the constructor that takes a `String` as its parameter, which is the string displayed in the window.

Source: http://www.toves.org/books/java/ch24-swing/index.html