
MENUS AND ACTIONS

The `JMenuBar` and `JMenu` classes in Swing work just about identically to those in the AWT. However, the `JMenuItem` class adds constructors that allow you to include an image alongside the menu text. To create a menu, you create a menu bar, add top-level menus and then add menu items to each of the top-level menus.

```
JMenuBar mbar = new JMenuBar();           //menu bar
setJMenuBar(mbar);                       //add to JFrame
JMenu mFile = new JMenu("File");         //top-level menu
mbar.add(mFile);                          //add to menu bar
JMenuItem Open = new JMenuItem("Open"); //menu items
JMenuItem Exit = new JMenuItem("Exit");
mFile.add(Open);                          //add to menu
mFile.addSeparator();                     //put in separator
mFile.add(Exit);
```

`JMenuItems` also generate `ActionEvents`, and thus menu clicks causes these events to be generated. As with buttons, you simply add action listeners to each of them.

```
Open.addActionListener(this);           //for example
Exit.addActionListener(this);
```

Action Objects

Menus and toolbars are really two ways of representing the same thing: a single click interface to initiate some program function. Swing also provides an `Action` interface that encompasses both.

```
public void putValue(String key, Object value);
public Object getValue(String key);
public void actionPerformed(ActionEvent e);
```

You can add this interface to an existing class or create a `JComponent` with these methods and use it as an object which you can add to either a `JMenu` or `JToolBar`. The most effective way is simply to extend the `AbstractAction` class. The `JMenu` and `JToolBar` will then display it as a menu item or a button respectively. Further, since an `Action` object has a single action listener built in, you can be sure that selecting either one will have exactly the same effect. In addition, disabling the `Action` object has the advantage of disabling both representations on the screen.

Let's see how this works. We can start with a basic abstract `ActionButton` class, and use a `Hashtable` to store and retrieve the properties.

```
public abstract class ActionButton extends AbstractAction
    implements Action
{
    Hashtable properties;

    public ActionButton(String caption, Icon img)
    {
        properties = new Hashtable();
        properties.put(DEFAULT, caption);
        properties.put(NAME, caption);
        properties.put(SHORT_DESCRIPTION, caption);
        properties.put(SMALL_ICON, img);
    }
    public void putValue(String key, Object value) {
        properties.put(key, value);
    }
    public Object getValue(String key) {
        return properties.get(key);
    }
    public abstract void actionPerformed(ActionEvent e);
}
```

The properties that `Action` objects recognize by key name are

- `String DEFAULT`
- `String LONG_DESCRIPTION`
- `String NAME`
- `String SHORT_DESCRIPTION`
- `String SMALL_ICON`

The `NAME` property determines the label for the menu item and the button, and in theory the `LONG_DESCRIPTION` should be used. This latter feature is not implemented in Swing 1.0x, but is expected to be in Java 1.2. The icon feature does work correctly.

Now we can easily derive an `ExitButton` from the `ActionButton` like this:

```
public class ExitButton extends ActionButton
{
    JFrame fr;
    public ExitButton(String caption, Icon img, JFrame frm) {
        super(caption, img);
    }
}
```

```

        fr = frm;
    }
    public void actionPerformed(ActionEvent e)    {
        System.exit(0);
    }
}

```

and similarly for the FileButton. We add these to the toolbar and menu as follows:

```

//Add File menu
JMenu mFile = new JMenu("File");
mbar.add(mFile);

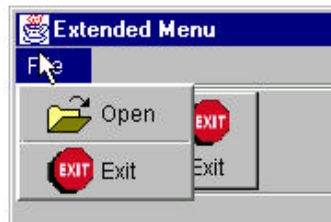
//create two Action Objects
Action Open = new FileButton("Open",
    new ImageIcon("open.gif"), this);
mFile.add(Open);

Action Exit = new ExitButton("Exit",
    new ImageIcon("exit.gif"), this);
mFile.addSeparator();
mFile.add(Exit);
//add same objects to the toolbar
toolbar = new JToolBar();
getContentPane().add(jp = new JPanel());
jp.setLayout(new BorderLayout());
jp.add("North", toolbar);

//add the two action objects
toolbar.add(Open);
toolbar.add(Exit);

```

This code produces the program window shown below:



the menu or the text on the toolbar. However, the *add* methods of the toolbar and menu have a unique feature when used to add an ACTION OBJECT. They return an object of type JButton or JMenuItem respectively. Then you can use these to set the features the way you want them. For the menu, we want to remove the icon

```

Action Open = new FileButton("Open",

```

```

        new ImageIcon("open.gif"), this);
menuItem = mFile.add(Open);
menuItem.setIcon(null);

```

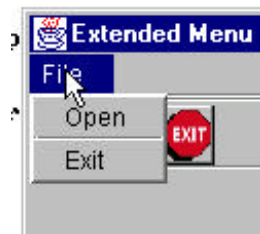
and for the button, we want to remove the text and add a tooltip:

```

JButton button = toolbar.add(act);
button.setText("");
button.setToolTipText(tip);
button.setMargin(new Insets(0,0,0,0));

```

This gives us the screen look we want:



Design Patterns in the Action Object

One reason to spend a little time discussing objects that implement the Action interface is that they exemplify at least two design patterns. First, each Action object must have its own *actionListener* method, and thus can directly launch the code to respond that that action. In addition, even though these Action objects may have two (or more) visual instantiations, they provide a single point that launches this code. This is an excellent example of the Command pattern.

In addition, the Action object takes on different visual aspects depending on whether it is added to a menu or to a toolbar. In fact you could decide that the Action object is a Factory pattern which produces a button or menu object depending on where it is added. In fact, it does seem to be a Factory, because the toolbar and menu *add* methods return instances on those objects. On the other hand, the Action object seems to be a single object, and gives different appearances depending on its environment. This is a description of the State pattern, where an object seems to change class (or methods) depending on the internal state of the object.

One of the interesting and challenging things about the design patterns we discuss in this book is that once you start looking at it, you discover that they are represented far more widely than you first assumed. In

some cases their application and implementation is obvious, and in other cases the implementation is a bit subtle. In these cases you sort of step back, tilt your head, squint and realize that from that angle it *looks* like a pattern where you hadn't noticed one before. Again, being able to label the code as exemplifying a pattern makes it easier for you to remember how it works and easier for you to communicate to others how that code is constructed.