Chapter 2: Menus, Menu Items, and the Menu Bar

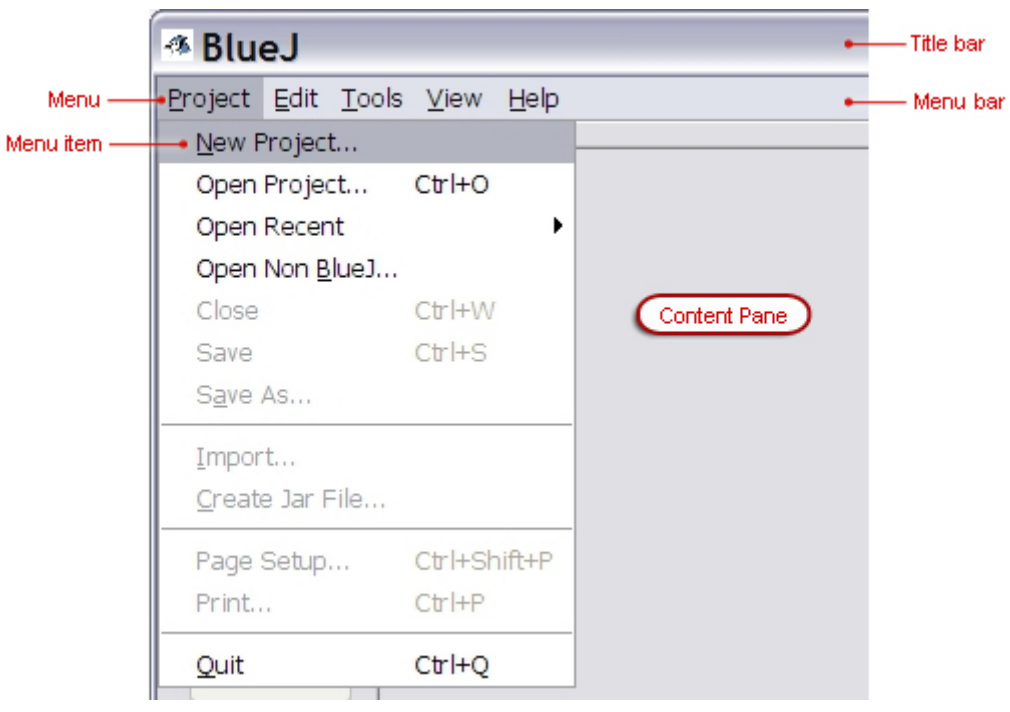
Menus, Menu Items, and the Menu Bar

Our next GUI step will add menus to our windows. We use three Swing classes to build menus: JMenuBar, JMenu, and JMenuItem.

The JMenuBar class represents a *menu bar* that shows up just below the title bar of our window. A window can only have one menu bar. For those of you working on a Mac, the default Java menu bar position will look strange because it's in the window instead of at the top of the screen. It takes an extra line of code in the main() method to put the menu bar at the top of the screen. I'll show it to you when we get to the code.

The JMenu class creates individual *menus* for the menu bar, like the usual File, Edit, and Help menus that are in many applications. You can use this class to add as many menus as you need to the menu bar or to add submenus to other menus. JMenus can also create pop-up menus, which we usually associate with right-clicks, but we won't deal with them in this lesson.

Last in the menu hierarchy is the JMenuItem class, which creates the individual *menu items* in each menu. A menu can contain as many items as you need, but if the menu gets too long, it makes more sense to use submenus than to have one long menu. Examples of menu items include the usual Open and Save menu items in the File menu. Just so we have our terminology straight, here's what I'm talking about:



GUI terminology

Normally, we add a menu bar to our frame first, then we create menus to add to the menu bar, and finally, we create menu items to add to the menus. Let's give it a try.

Here is a bare-bones GUI window program that we will add menus to. We'll start with this example for a couple of lessons, and we'll later go on to add other features based on our menu options. But for now, here's the starting point for this lesson:

```
import java.awt.*;

import java.awt.event.*;

import javax.swing.*;


public class GUIMenu

{

    private JFrame frame;


    public static void main (String[] args)

    {

      GUIMenu gui = new GUIMenu();

      gui.start();

    }


    public void start()

    {

      frame = new JFrame("GUI Menus");

      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      Container contentPane = frame.getContentPane();


      frame.setSize(300, 300);

      frame.setVisible(true);

    }

}
```

If you run this program, you will see yet one more empty window. That's getting pretty boring, isn't it? So let's do something to improve it. Let's add menus!

Just to keep our program better organized, let's add a method that will build our menu for us. Since it will only be called internally, we'll make it a private method and name it *makeMenus()*. Our start() method will call the new method, so let's add the call first, then we'll write the method. Here is the call, which we'll put after the line that gets our content pane:

```
makeMenus();
```

That was pretty simple, wasn't it? Now let's create our new method at the end of the class, just before the last closing bracket (}), like this:

```
private void makeMenus()

{

}
```

We already know we'll need a JMenuBar object, some JMenu objects, and some JMenuItem objects. So let's create some names in our new method that will let us refer to those objects, like this:

```
JMenuBar menuBar;

JMenu menu;

JMenuItem menuItem;
```

Now that we have some names, let's put them to work. First, we need to create our menu bar object and attach it to our frame. That takes two statements that look like this:

```
menuBar = new JMenuBar();

frame.setJMenuBar(menuBar);
```

The first line creates a new JMenuBar object. The second line uses the frame's setJMenuBar() method to attach the menu bar to the frame. As I mentioned earlier, a window can only have one menu bar, so we've just used up our limit.

If you run the program now, it won't look much different than it did before. If you look closely, you might see a small, almost invisible menu bar across the top of the frame. Since we haven't added any menus yet, there's nothing to show in it. So let's fix that little problem. These next two lines show you how to create a menu and add it to the menu bar:

```
menu = new JMenu("File");

menuBar.add(menu);
```

The first of these lines creates our first menu and gives it the label "File." The second line adds the menu to our menu bar. Menus will appear in the menu bar in the order we add them, from left to right. If you run the program now, it will look like this:

A window with a menu

Note for the Mac Java programmers out there: If you want the menu bar to jump to the top of the screen like a Mac menu should, add this line to your main() method:

```
System.setProperty("apple.laf.useScreenMenuBar", "true");
```

That line will set Java's System property that controls where menu bars appear on a Mac. It won't work anywhere but on a Mac. Now, back to the main topic.

Our application is starting to look more like the windows we're used to seeing. You can see the menu bar at the top of the window and the File menu at its left edge. But if you click the menu, nothing happens yet because there is nothing in the menu to display. Let's add some menu items to it, like this:

```
menuItem = new JMenuItem("New");

menu.add(menuItem);

menuItem = new JMenuItem("Open.");

menu.add(menuItem);

menuItem = new JMenuItem("Save");

menu.add(menuItem);

menuItem = new JMenuItem("Save As.");

menu.add(menuItem);

menu.addSeparator();

menuItem = new JMenuItem("Exit");

menu.add(menuItem);
```

You can see that creating these menu items is very similar to creating the menu. We just create a JMenuItem object and give its constructor the text we want to appear in the menu. Then we add the menu item to the menu.

Take a look at the third-to-last line in the code above: `menu.addSeparator();`. This command is unique to the JMenu class. It adds a separator line to the menu so we can group related items together. In this case, we added it between Save As . . . and Exit to separate the file processing menu items from the Exit item. If you run the program now and click **File**, you should see something like this:

A menu with items

In case you're wondering, you can add another menu to an existing menu, too. If you add a menu to a menu, it creates a submenu that opens when you click its name in the first menu. You can layer menus as deep as needed, one inside another. Since you already have all the tools to do that, I'm going to leave it for you to try as an exercise in the assignment.

Now, as you will find out if you click any of the menu items in our File menu, none of them actually do anything yet. Believe it or not, you already know how to make them work. We make them work the same way we did buttons: with ActionListeners. We will add an ActionListener object to each menu item to "listen" for a mouse click.

We'll organize our menu actions a little differently than we did our button actions in the last lesson, though. Since an application can have any number of menus and menu items, putting all their actions into one ActionPerformed() method would make that method very long. It would also violate the good coding practice of only asking a method to do one thing. So we will perform each menu item's action in a separate method, and our ActionPerformed() method will call them when the time is right. Not only will that organize our program better, but it will make future changes to the program much easier to manage. (And we *will* be changing it!)