Chapter 4: Inner Classes

Inner Classes

We have one more topic for today. It relates to the GUI menu program we just wrote, and more specifically, to cleaning up the actionPerformed() method. You might wonder why it needs cleaning up. After all, it works, doesn't it?

Yes, it works, but it has some drawbacks. First, we removed the menu actions from the actionPerformed method to keep it from having to do a lot of different things, since that's not very good programming practice. But we still have a bunch of different branches in it that require a bunch of *if . . . else if . . .* logic. Wouldn't it be nice to be able to just get rid of all that?

Second, the *if* statements depend on menu item text. That's not very good practice, either. What if we wanted to change the text on a menu item? We would have to remember to change it in two places. Worse yet, what if we wanted to translate our application into another language? We'd have to change *all* the menu items in both places. It just gives us more chances to make mistakes.

Last, what we have is a menu listener starting a method that does nothing but call another method. That seems like a waste of a step. It makes more sense to have the listener call the second method directly, doesn't it?

We're going to settle all these issues with what Java calls *inner classes*. An inner class is a class that is defined inside another class called (naturally) the *outer class*. We can create a listener class inside our GUIMenu class. In fact, we can create any number of listener classes in there—one for each menu item, even—and each can manage its own action directly. That probably doesn't make much sense yet, so let me show you what I mean. Here's an example:

```
public class GUIMenu
{
    // . . .
    menuItem = new JMenuItem("New");
    menuItem.addActionListener(new newListener());
    menu.add(menuItem);
    // . . .
    private class newListener implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            JOptionPane.showMessageDialog(frame,
                    "The  File > New  menu option was clicked",
                    "Menu Item Click",
                    JOptionPane.INFORMATION_MESSAGE);
        }
    }
    // . . .
}
```

There are several things I want you to notice about this example. First, our outer class, GUIMenu, is no longer an ActionListener and does not need to have `implements ActionListener` in its first line.

Next, when we add our listener to a menu, we don't add the outer class as the listener. Instead, we add a new object of a listener class specifically designed to listen to this menu item. Each menu item will have its own dedicated listener class to handle its action.

Also note that we don't have an actionPerformed() method in our outer GUIMenu class any longer. We don't need it since each menu item now has its own listener. That means we were also able to get rid of that long set of *if* statements that we needed to manage all the calls to the different menu action methods.

And last, you can see that we don't have a separate method defined for each menu item anymore. Instead, each menu item has its own inner class defined as an ActionListener, with its own actionPerformed() method, which defines the action to take for that specific menu item. While this can generate a lot of inner classes if a window has lots of menu items or other interactive elements, it's a more structured and object-oriented approach and is generally preferable to the first one we tried.

Give that technique a try. Go ahead and add separate listener classes for each menu item, then add an object of the correct listener type to each item. When you run the program, it shouldn't look or act any different than it did before. If you have any problems making it work, you can see my version here, but please try to set it up yourself before you look.

Solution: Try not to peek!

Before we move on, I'd like to pass along a little more information about inner classes:

- Just as a matter of style, inner classes are usually placed at the end of the outer class, after all its methods.
- Inner classes, since they are internal to another class, have access to all the data and methods in the outer class, even private elements. In that way, they're like methods. Notice, for example, that in our inner class above we used an outer class variable, frame, without any difficulty. Outer class methods can also be called from within an inner class.
- Instances of inner classes are attached to instances of the outer class. The inner class instances do not exist independently and cannot be referenced independently of outer class instances.
- Some programmers, when they only use an inner class one time, use a technique called *anonymous inner classes*. That technique goes one step further and defines inner classes without names right where they are used. In the example we wrote in this lesson, that would mean defining our five inner classes right in the argument list for the addActionListener() method. I prefer the method we used in this lesson because I think it's less complicated and easier to read. We won't be using anonymous inner classes in this course, but I wanted you to be familiar with the term in case you come across it.

If you would like more information on inner classes, there are a couple of links in the Supplementary Material section that will show you other ways to use them.

Menu Mnemonics and Accelerators

Go ahead, try saying "menu mnemonics" five times fast! Just kidding. Our last topic for today is how to set up keyboard shortcuts for menus and menu items. A *mnemonic* for a menu is a key that, in combination with the ALT key, will make the menu act just like you clicked the associated menu or item. In order for a mnemonic to work, the associated menu or item must be visible. An example of using mnemonics would be hitting ALT + F to pull down the File menu, then typing O to select the Open menu item.

There are also *menu accelerators* that will bypass the menu hierarchy and act like the menu item was clicked, whether or not the item was visible. For example, you could type CTRL + S to save, whether or not the Save menu item is visible.

Adding a mnemonic to a menu or a menu item is easy. Both types of objects have a method named setMnemonic() that takes one argument: an integer designating the key that will be the mnemonic. Calling the method not only sets the mnemonic, but it underlines the first occurrence of that letter in the menu as a visual cue. We can set the mnemonic for the File menu and the Open menu item like this:

```
menu.setMnemonic(KeyEvent.VK_F);

menuItem.setMnemonic(KeyEvent.VK_O);
```

The values for the argument are available in a class called *KeyEvent*, so we don't even have to worry about what key is what value. If you look at the KeyEvent class in the Java API, you'll see 100 or so key values, one for every possible key on the keyboard. For example, VK_A is the value for the A key, VK_Z is the value for the Z key, VK_TAB is the TAB key, VK_F12 is the F12 key, and so on.

As I said above, the setMnemonic() method automatically underlines the first instance of the letter you choose for the menu item. But what if you want to underline the second instance of the letter? For example, in the Save As . . . menu, the setMnemonic() method would normally underline the first A, the one in the word *Save*. But most applications underline the second A instead, the one in *As*. We can use the method setDisplayedMnemonicIndex() to change it, like this:

```
menuItem.setDisplayedMnemonicIndex(5);
```

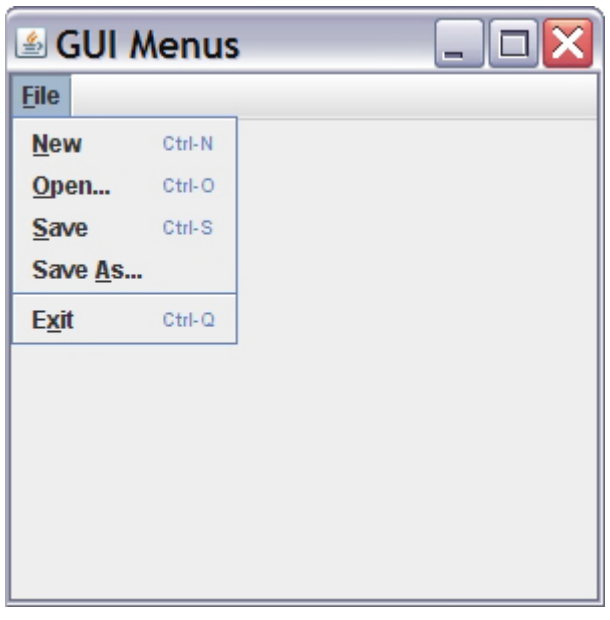
The first letter in the menu title is position zero, the second is position one, and if you keep counting you'll see that the A in *As* is position five. The call above underlines the letter in that position, so the menu shows up with the title Save As . . . , which is just the way we want it.

Unlike menu mnemonics, accelerator keys *only* work for menu items, not for menus. They are also set using a method call, this time setAccelerator(). Here is what this method call for the Save menu item looks like:

```
menuItem.setAccelerator(
        KeyStroke.getKeyStroke(KeyEvent.VK_S,
                               Event.CTRL_MASK));
```

This call uses a method called getKeyStroke(), from the KeyStroke class, to simulate a keystroke involving two keys. This call tells Java that the accelerator key for the Save menu item is the letter S modified (or *masked*) by the CTRL key, which we normally write CTRL + S.

As our last exercise for this lesson, go ahead and set up the mnemonics ALT + N, ALT + O, ALT + S, ALT + A, and ALT + X for the New, Open, Save, Save As, and Exit menu items, respectively, in addition to the mnemonic ALT + F for the File menu. Also set up the accelerator keys CTRL + N, CTRL + O, CTRL + S, and CTRL + Q for the New, Open, Save, and Exit menu items. If you run the program after getting that set up and pull down the menu, it should look like this:



Menu with mnemonics and accelerators

If you have trouble, you can let me know in the Discussion Area. You can also compare your code to mine, which is here.

Solution: Try not to peek!