**Q:**So what are inner classes good for anyway?

**A:**Inner classes nest within other classes. A normal class is a direct member of a package, a top-level class. Inner classes, which became available with Java 1.1, come in four flavors:

* Static member classes
* Member classes
* Local classes
* Anonymous classes

Let's take a quick look at each in turn.

Briefly, a *static member class* is a static member of a class. Like any other static method, a static member class has access to all static methods of the parent, or top-level, class.

Like a static member class, a *member class* is also defined as a member of a class. Unlike the static variety, the member class is instance specific and has access to any and all methods and members, even the parent's this reference.

*Local* classes are declared within a block of code and are visible only within that block, just as any other method variable.

Finally, an *anonymous* class is a local class that has no name.

To answer your specific question, I'll focus on the member and anonymous inner classes since those are the ones you'll likely encounter and use. To me, the advantages of inner classes can be divided into three categories: an object-oriented advantage, an organizational advantage, and a call-back advantage.

**The object-oriented advantage**

In my humble opinion, the most important feature of the inner class is that it allows you to turn things into objects that you normally wouldn't turn into objects. That allows your code to be even more object-oriented than it would be without inner classes.

Let's look at the member class. Since its instance is a member of its parent instance, it has access to every member and method in the parent. At first glance, this might not seem like much; we already have that sort of access from within a method in the parent class. However, the member class allows us to take logic out of the parent and objectify it. For example, a tree class may have a method and many helper methods that perform a search or walk of the tree. From an object-oriented point of view, the tree is a tree, not a search algorithm. However, you need intimate knowledge of the tree's data structures to accomplish a search.

An inner class allows us to remove that logic and place it into its own class. So from an object-oriented point of view, we've taken functionality out of where it doesn't belong and have put it into its own class. Through the use of an inner class, we have successfully decoupled the search algorithm from the tree. Now, to change the search algorithm, we can simply swap in a new class. I could go on, but that opens up our code to many of the advantages provided by object-oriented techniques.

**The organizational advantage**

Object-oriented design isn't everyone's thing, but luckily, inner classes provide more. From an organizational point of view, inner classes allow us to further organize our package structure through the use of namespaces. Instead of dumping everything in a flat package, classes can be further nested within classes. Explicitly, without inner classes, we were limited to the following hierarchy structure:

package1

**class** 1

**class** 2

...

**class** n

...

**package** n

With inner classes we can do the following:

**package** 1

**class** 1

**class** 2

**class** 1

**class** 2

...

**class** n

Used carefully, inner classes can provide a structural hierarchy that more naturally fits your classes.

**The callback advantage**

Inner member classes and anonymous classes both provide a convenient method for defining callbacks. The most obvious example relates to GUI code. However, the application of the callback can extend to many domains.

Most Java GUIs have some kind of component that instigates an actionPerformed() method call. Unfortunately, most developers simply have their main window implement ActionListener. As a result, all components share the same actionPerformed() method. To figure out which component performed the action, there is normally a giant, ugly switch in the actionPerformed() method.

Here's an example of a monolithic implementation:

**public** **class** **SomeGUI** **extends** **JFrame** **implements** **ActionListener**

{

**protected** **JButton** button1;

**protected** **JButton** button2;

...

**protected** **JButton** buttonN;

**public** **void** actionPerformed(**ActionEvent** e)

{

**if**(e.getSource()==button1)

{

*// do something*

}

**else** **if**(e.getSource()==button2)

{

... you **get** the picture

Whenever you see switches or large if/if else blocks, loud alarm bells should begin to ring in your mind. In general, such constructs are bad object-oriented design since a change in one section of the code may require a corresponding change in the switch statement. Inner member classes and anonymous classes allow us to get away from the switched actionPerformed() method.

Instead, we can define an inner class that implements ActionListener for each component to which we want to listen. That may result in many inner classes. However, we can avoid large switch statements and have the added bonus of encapsulating our action logic. Moreover, that approach may improve performance. In a switch where there are *n* comparisons, we can expect *n/2*comparisons in the average case. Inner classes allow us to set up a 1:1 correspondence between the action performer and the action listener. In a large GUI, such optimizations can make a substantial impact on performance. An anonymous approach may look like this:

**public** **class** **SomeGUI** **extends** **JFrame**

{

... button member declarations ...

**protected** **void** buildGUI()

{

button1 = **new** **JButton**();

button2 = **new** **JButton**();

...

button1.addActionListener(

**new** java.awt.**event**.**ActionListener**()

{

**public** **void** actionPerformed(java.awt.**event**.**ActionEvent** e)

{

*// do something*

}

}

);

.. repeat **for** each button

Using inner member classes, the same program would look like this:

**public** **class** **SomeGUI** **extends** **JFrame**

{

... button member declarations

*// inner class definitions*

**class** **Button1Handler** **implements** **ActionListener**

{

**public** **void** actionPerformed(**ActionEvent** e)

{

*// do something*

}

}

... define an inner member **class** **for** each button

**protected** **void** buildGUI()

{

*// initialize the buttons*

button1 = **new** **JButton**();

button2 = **new** **JButton**();

...

*// register an inner class action listener instance*

*// for each button*

button1.addActionListener(**new** **Button1Handler**());

.. repeat **for** each button

Since inner classes have access to everything in the parent, we can move any logic that would have appeared in a monolithic actionPerformed()implementation to an inner class.

I prefer to use member classes as callbacks. However, that is a matter of personal preference. I just feel that too many anonymous classes clutter code. I also feel that anonymous classes can become unwieldy if they are larger than one or two lines.

**Disadvantages?**

As with anything else, you have to take the good with the bad. Inner classes have their disadvantages. From a maintenance point of view, inexperienced Java developers may find the inner class difficult to understand. The use of inner classes will also increase the total number of classes in your code. Moreover, from a development point of view, most Java tools come up a bit short on their support of inner classes. For example, I use IBM's VisualAge for Java for my day-to-day coding. While inner classes will compile within VisualAge, there is no inner class browser or template. Instead, you must simply type the inner class directly into the class definition. That unfortunately makes browsing the inner class difficult. It is also difficult to type since you lose many of VisualAge's code completion aids when you type into the class definition or use an inner class.