



Practical Use of Java Inner Classes

By [Joseph H McIntyre](#)
IBM Personal Software Products

Overview

Inner classes are a new feature of version 1.1 of the Java language. Inner classes simplify programming constructs for organizing class definitions and event handling code, resulting in easier-to-read and -manage source code. A special type of inner class, the anonymous inner class, can implement an interface directly.

Opportunities abound for using this new technique, and overuse can be tempting. Examples show not only how to use inner classes, but where to use alternative techniques as well.

This paper presents inner classes from two perspectives: 1) using the language to implement inner classes directly and 2) implementing the equivalent code using standard Java syntax (syntax that may be used with older versions of Java or with compilers that do not support inner class syntax).

Boldface will be used to highlight new or changed lines of code when one of the following examples builds on a preceding example.

Defining Inner Classes

The primary benefits of inner classes are realized when working with classes that are related to each other. The classes may be related by organization, such as in an address class that is part of the definition of a customer class, or by usage, such as in Java programs that use the new event model introduced with Java 1.1. In both of these cases, Java developers could end up with many small Java source files, and they would undoubtedly want to place all the classes within one file.

The Java language specifies that each public class must be in its own source file. Inner classes provide a solution for this inconvenience by allowing classes to be defined with public access within the scope of their containing class. In the example of a linked list, this allows the Node class to be defined within the LinkedList class and still be accessed by an external class. The following example shows an excerpt from a class definition for a linked list:

```

public class LinkedList
{
    Node first;

    public class Node
    {
        Object data;
        Node next;
    }
}

```

An application using the `LinkedList` would be able to create new nodes and add them to the list and to access the values in the node, even though the class `Node` is not defined in its own file. Without inner classes, the class `Node` would be defined in its own source file, `Node.java`. This example shows the value of using inner classes to organize related classes:

A new event handling model was introduced with Java 1.1 that is based on an event source/listener model. The listener has methods invoked when the event source to which it is listening fires an event.

The following example provides a short introduction to event handling in Java 1.1. It shows the relationship between a button and a dialog and the activity related to setting up the event notification and the firing of an event.

```

Button      <-- 1. addActionListener      Dialog
Event source 2. actionPerformed  --> Event listener

```

After a `Button` is created, an `ActionListener` is created by the `Dialog` and attached to the `Button` instance. The `Button` instance records the interest of the dialog in receiving Action events. When an action occurs on the button, such as clicking, each of the attached listeners is notified of the action through the `actionPerformed` method, which is defined in the `ActionListener` interface. The `Dialog` class must implement the `ActionListener` interface or have an inner class that implements it.

The new event model is very useful; however, there are two basic problems. Each set of events that a listening object is interested in requires that the interface for the listener be implemented in the listening object. While this is trivial for events like `ActionListener`, which only contain one method to implement, it becomes unwieldy when listener interfaces that contain many methods, such as `WindowListener` and `MouseListener`, are used.

The other basic issue is the placement of the code that responds to the events. In many cases, it is desirable to place the code that responds to the events near the definition of the object to which the listener is attached. Using inner classes, the code that handles the events an object receives may be placed inline with the method that attaches the listener to the object.

"An Example of a Simple Event" (later in this paper) describes how to simplify the implementation of event handling code.

Using a Named Inner Class

A common case for a named inner class is the embedded data structure, such as

```

public class ShipInstruction
{
    public class ShipAddress
    {
        String addressLabel;
        int routingNumber;
    }

    int documentNumber;
}

```

Compiling this class will create the class files `ShipInstruction.class` and `ShipInstruction$ShipAddress.class`. Like regular classes, a Java compiler generates a class file for each inner class it encounters. However, the name of the class file reflects the context of its definition. The class file name consists of the outer class name followed by the dollar sign followed by the name of the inner class.

This use of inner classes enhances the readability and ease of understanding of the data structures, since they can be viewed together in the same class definition. It also allows for the placement of multiple public classes in one source file, reducing the number of small source files that lack meaning outside the context of an enclosing or related class. While there can only be one public outer class, there may be multiple public inner classes defined within the outer class.

Inner classes also have a special relationship with their enclosing classes for attribute access. The following example shows an attribute defined in an inner class being initialized using the value of a private attribute from the outer class. If the class `t2` were defined in its own file, it would not be able to access the private attribute `attr1` in class `t1`. However, an inner class has access to the private attributes of its enclosing class or classes.

```

public class t1
{
    private int attr1;

    public class t2
    {
        private int attr2;

        public t2 ()
        {
            attr2 = attr1;
            System.out.println ("attr2 = " + attr2);
        }
    }

    public t1 ()
    {
        attr1 = 5;
        new t2 ();
    }

    public static void main (String args[])
    {
        new t1 ();
    }
}

```

```
    }
}
```

The result of running this sample is the text "attr2 = 5".

When referencing an inner class, the type of the inner class is defined using the outer class as a qualifier. Using the ShipInstruction example above, an application referencing the ShipAddress class would use the name ShipInstruction.ShipAddress.

When creating instances of an inner class, the new instance of the inner class must be created from an existing instance of the outer class. An instance is required because the inner class has an implicit "this" pointer to the instance of the outer class in which it is created. The "this" pointer is a special value that contains a reference to the current instance of a class. The following example shows how to create the class instances to prepare them for accepting attribute assignments:

```
// create outer class instance
ShipInstruction si = new ShipInstruction ();

// create inner class using outer class instance
ShipInstruction.ShipAddress sa = si. new ShipAddress ();

// assign values to attributes
si.documentNumber = 1000;
sa.routingNumber = 27;
```

Note the following about this example:

- Creating an instance of ShipInstruction does not create an instance of ShipAddress.
- An instance of ShipAddress must be created from an instance of ShipInstruction; it cannot be created independently of a ShipInstruction instance.
- The syntax for creating an instance of an inner class is:

```
<outer class instance>. new <inner class name> (<constructor parameters>);
```

As shown, named inner classes are very useful in organizing classes. In the next section, the anonymous inner class is introduced.

Using an Anonymous Inner Class

The anonymous inner class allows code to be placed inline where an object parameter is expected. In many cases, event handling logic is well suited for this use. An anonymous inner class specifies only the name of the class it extends or interface it implements; it does not have a name specified for itself. When the compiler encounters an anonymous inner class, it will generate a name consisting of the name of the class being compiled followed by a dollar sign and a digit.

Note that an anonymous class may implement an interface; this is not allowed according to the Java language specification. However, since a class that extends Object is being created under-the-covers by the compiler, this rule is not violated. This

is the only case in which an interface may be implemented.

An example of a button event being captured demonstrates the usefulness of the anonymous inner class: Line 5: The button is created in the constructor. Line 6: An action listener is attached to the button; however, instead of placing this or an ActionListener instance variable as the parameter, the code for the ActionListener is placed inline. Lines 7-12: The implementation of the listener is coded inline. Line 12: The inline code is the parameter to the addActionListener method; therefore, the closing paren is added after the closing brace.

```

1  class MyClass
2  {
3      public MyClass ()
4      {
5          Button b = new Button ("Ok");
6          b.addActionListener (new ActionListener ()
7              {
8                  public void actionPerformed (ActionEvent event)
9                  {
10                     ...
11                 }
12             });
13     }
14 }
```

Note the following from the above example:

- ActionListener is an interface, not a class. The anonymous inner class will map to "class xxx extends" or "class xxx implements," depending on whether a class or interface is specified.
- The new clause creates the class with the appropriate constructor. Note that if an interface is being implemented, the parameter list is always empty, calling the default constructor of Object.
- The code for implementing the interface or extending the class is placed inline within braces.
- After the closing brace, the closing paren and semicolon for the addActionListener method call are added.

The following sections show examples of using and simulating inner classes.

An Example of a Simple Event

Inner classes are often used for handling for Java 1.1.x events. For example, any class that derives from java.awt.Frame will typically act on the window close event. Using an anonymous inner class, the window close event may be received by the frame by using the following code:

```

import java.awt.*;
import java.awt.event.*;

public class MyFrame extends Frame
{
```

```

public MyFrame ()
{
    // listen for window events
    addWindowListener (new WindowAdapter ()
    {
        // on closing event, terminate application
        public void windowClosing (WindowEvent event)
        {
            System.exit (0);
        }
    });

    setBounds (100, 100, 300, 200);
    show ();
}

public static void main (String args[])
{
    new MyFrame ();
}
}

```

The above example displays an empty frame window that will terminate the application when the close action is initiated by the user. In the addWindowListener method, an anonymous inner class is created that extends the WindowAdapter class and implements the windowClosing method.

An equivalent example (below) may be created without using an anonymous inner class by writing a separate class and passing an instance of the class to the addWindowListener method. Comparing these two approaches highlights the advantages of using the inner class feature: 1) improved readability and 2) placement of the code that performs the action at the location where the event listener is created.

```

import java.awt.*;
import java.awt.event.*;

// class listens for window closing event
class MyFrame_1 extends WindowAdapter
{
    // on closing event, terminate application
    public void windowClosing (WindowEvent event)
    {
        System.exit (0);
    }
}

public class MyFrame extends Frame
{
    public MyFrame ()
    {
        // sent window events to MyFrame_1
        addWindowListener (new MyFrame_1 ());

        setBounds (100, 100, 300, 200);
        show ();
    }

    public static void main (String args[])

```

```

    {
        new MyFrame ();
    }
}

```

When these two examples are compiled, the only differences are the file names for the generated class files. The first example, using the inner class feature, will create the files `MyFrame.class` and `MyFrame$1.class`. The second example, simulating the function of inner classes, will create the files `MyFrame.class` and `MyFrame_1.class`. The function of the two examples will be exactly the same.

A benefit of using the second example implementation is the ability to reuse the event class elsewhere. In the first example, the anonymous inner class is hidden from other classes. In the second example, the simulated inner class (`MyFrame_1`) could be generalized and used in other applications in which the equivalent event handling function is required.

To decide which implementation technique to use, determine whether the function being implemented will be either used elsewhere in this project or used commonly in other projects. If it will be, create a separate class and consider adding it to a utility package. Otherwise, take advantage of the inner class feature.

Calling an Outer Class Method from an Inner Class

Many Java programs store their state on exit. The following example extends the previous inner class example by adding a method call to the event handling code that calls a method defined in the outer class:

```

import java.awt.*;
import java.awt.event.*;

public class MyFrame extends Frame
{
    public MyFrame ()
    {
        // extend WindowAdapter class with anonymous inner class
        addWindowListener (new WindowAdapter ()
        {
            public void windowClosing (WindowEvent event)
            {
                // call outer class method
                saveState ();
                System.exit (0);
            }
        });

        setBounds (100, 100, 300, 200);
        show ();
    }

    // Note that this method may be private; however, it appears
    // that a compiler bug precludes this when the method is
    // called from within an anonymous inner class.
    void saveState ()
    {
    }
}

```

```

    public static void main (String args[])
    {
        new MyFrame ();
    }
}

```

If the above example is to be simulated without using an inner class, the additional class defined requires a reference to the outer class instance to make the `saveState` method call. Note that the additional class is a separate class and is, therefore, subject to the normal Java language scoping rules. Therefore, the scope of the method being called in the outer class must be `package` (the default scope when no qualifier is specified) or `public` in order for the simulated inner class to see it.

Building on the non-inner class example from the previous section, the simulated inner class is expanded to hold a reference to the outer class:

```

import java.awt.*;
import java.awt.event.*;

class MyFrame_1 extends WindowAdapter
{
    // reference to outer class instance
    MyFrame object;

    // constructor, store reference to outer class instance
    public MyFrame_1 (MyFrame object)
    {
        this.object = object;
    }

    public void windowClosing (WindowEvent event)
    {
        // call method in outer class using stored reference
        object.saveState ();
        System.exit (0);
    }
}

public class MyFrame extends Frame
{
    public MyFrame ()
    {
        // pass this reference to simulated inner class
        addWindowListener (new MyFrame_1 (this));

        setBounds (100, 100, 300, 200);
        show ();
    }

    void saveState ()
    {
    }

    public static void main (String args[])
    {
        new MyFrame ();
    }
}

```


When developing classes using a compiler that does not support inner classes, the above example may be used as a standard template for creating classes that simulate the capabilities of inner classes for event handling.

For examples in which instance information, in addition to the object reference, is required, such as the example in the section that follows, the constructor for `MyFrame_1` would add parameters for each of the instance variables. These parameters are unnecessary when inner classes are used.

Using Multiple Inner Classes

Many opportunities for using multiple inner classes will arise. In some cases, this will be the best course to follow, but in other cases, it is better to use an alternative technique.

The following example is typical of graphical applications, with multiple buttons presented with which the user can interact. An inner class may be used to associate logic with each button instance.

Using this approach has tradeoffs. One class file will be generated for each inner class defined, and code cannot be shared between two buttons unless another method is added to the outer class that each inner class calls. For some cases, this tradeoff is justified; for other cases, there are alternative approaches to consider.

This first code fragment shows the use of inner classes to respond to events generated for all three buttons:

```
import java.awt.*;
import java.awt.event.*;

public class MyFrame extends Frame
{
    public MyFrame ()
    {
        // create buttons and anonymous handler classes to handle events
        Button b1 = new Button ("Ok");
        b1.addActionListener (new ActionListener ()
        {
            public void actionPerformed (ActionEvent event)
            {
                // code
            }
        });

        Button b2 = new Button ("Apply");
        b2.addActionListener (new ActionListener ()
        {
            public void actionPerformed (ActionEvent event)
            {
                // code
            }
        });

        Button b3 = new Button ("Cancel");
        b3.addActionListener (new ActionListener ()
```

```

        {
            public void actionPerformed (ActionEvent event)
            {
                // code
            }
        });
    ... code ...
}

```

This second code fragment shows how to implement the above example without inner classes, instead implementing the event listener as part of the definition of the class. In this example, this does not impact readability significantly but does require attributes to be defined and an if/else structure be implemented to differentiate the actions based on the button generating the event.

```

import java.awt.*;
import java.awt.event.*;

public class MyFrame extends Frame implements ActionListener
{
    Button b1;
    Button b2;
    Button b3;

    public MyFrame ()
    {
        // create buttons, point to self as event handler
        b1 = new Button ("Ok");
        b1.addActionListener (this);

        b2 = new Button ("Apply");
        b2.addActionListener (this);

        b3 = new Button ("Cancel");
        b3.addActionListener (this);

        ... code ...
    }

    public void actionPerformed (ActionEvent event)
    {
        // determine source from event information
        Object target = event.getSource ();
        if (target instanceof Button)
        {
            Button selected = (Button) event.getSource ();
            if (selected == b1)
            {
                // code Ok
            }
            else if (selected == b2)
            {
                // code Apply
            }
            else if (selected == b3)
            {
                // code Cancel
            }
        }
    }
}

```

```

    }
  }
}

```

And finally, this third example combines the two techniques using a single simulated inner class. In this example, a single additional class is created and all logic is contained within this external class. This removes the additional code from the application class and allows the use of constants to define the source of the action.

```

import java.awt.*;
import java.awt.event.*;

class MyFrame_1 implements ActionListener
{
    MyFrame object;
    int id;

    // constructor, store outer class object reference and action identifier
    public MyFrame_1 (MyFrame object, int id)
    {
        this.object = object;
        this.id = id;
    }

    public void actionPerformed (ActionEvent event)
    {
        // base action on stored action identifier in this instance
        switch (id)
        {
            case MyFrame.ID_OK:
                // code Ok
                break;

            case MyFrame.ID_APPLY:
                // code Apply
                break;

            case MyFrame.ID_CANCEL:
                // code Cancel
                break;
        }
    }
}

public class MyFrame extends Frame
{
    static final int ID_OK = 0;
    static final int ID_APPLY = 1;
    static final int ID_CANCEL = 2;

    public MyFrame ()
    {
        // create button and handler instance for Ok button
        Button b1 = new Button ("Ok");
        b1.addActionListener (new MyFrame_1 (this, ID_OK));

        // create button and handler instance for Apply button
        Button b2 = new Button ("Apply");
    }
}

```

```

        b2.addActionListener (new MyFrame_1 (this, ID_APPLY));

        // create button and handler instance for Cancel button
        Button b3 = new Button ("Cancel");
        b3.addActionListener (new MyFrame_1 (this, ID_CANCEL));

        ... code ...
    }
}

```

The above solution is very common for handling menus and button sets. Each of the examples shows a different approach for handling this problem, and each may be useful for different situations.

The first example, using anonymous inner classes, is useful when there are few items. Since there are only a few code sections placed inline, the size of the constructor will not be too long. Using inner classes is also suitable when a several types of events are handled, each type for only a few objects. For instance, if a class contained buttons, lists, and text fields, then three different interfaces would have to be implemented. Using inner classes and isolating the event handling code for each widget would be a better solution than creating three extra classes or implementing the many additional methods in the outer class.

The second example, which implements the interface with the outer class, is useful when the outer class is not too large and methods that must be implemented for the support interfaces are few. For an `ActionListener`, which only requires one method to be implemented, this is a good solution. For other interfaces, such as `WindowListener`, which define a large set of methods, the use of an inner class extending the corresponding Adapter class is usually the best choice. In the case of `WindowListener`, the `WindowAdapter` class would be extended.

The third example combines the approaches from the first two examples. It uses an external class to isolate the event handling code from the outer class, but it requires an additional class for each interface handled. This approach is useful when the outer class is large and there are few interfaces to handle or when the implementation of the event handling code may be reused. When reuse is possible, the event handling class may be placed in its own public class and used by different classes in the same manner as this third example demonstrates.

These three examples provide insight into the decisions involved in selecting the best way to handle events. As you can see, inner classes are a valuable addition to the list of alternatives. When implementing your classes, consider the ability to read and understand the code as one of the criteria in selecting the best approach for writing your event handling code.

Applying Inner Classes to Older Compilers

Many Java compilers, including the one shipped in version 1.0 of VisualAge for Java, do not support the syntax for inner classes. However, this does not prevent you from gaining the benefits of inner classes by using the techniques demonstrated for simulating inner classes using standard Java syntax. When you decide that using inner classes is the best implementation technique, use the equivalent simulated class instead. When the compiler is updated to support inner class syntax, you can cut and paste the code from the simulated class into the primary class with no change in logic and with minimal changes in the written code. This technique provides a stepping stone

that allows you to use inner class design and make implementation decisions today and prepares you for complete implementation as the tools are released.

Particularly in code reuse, especially for event-related activities, the use of simulated inner classes has been very useful in identifying candidate classes to incorporate into utility libraries. By following the naming technique (using an underscore and digit following the class name), it is easy to identify these classes. As your applications grow, create a utility package that contains these reusable classes. The most obvious utility classes are common event handling routines for GUI widgets; however, other categories of classes, such as business object interactions, are also good candidates.

Final Thoughts

Inner classes are indispensable in Java programming, whether they are implemented directly or by simulation. Opportunities for using inner classes can be inviting, but it is important to consider where they are used.

Use inner classes when

- few objects are used
- several interfaces are being implemented, with only a few objects implementing each interface
- code in a large class is easier to read by placing the event code near the object to which it corresponds

Don't use inner classes when

- the code contained within an anonymous inner class is useful somewhere else. If so, place it in a regular class, since code in an anonymous inner class cannot be accessed from another class.
- the use of inner classes results in unnecessary complexity or difficult reading. As the event handling scenario shows, alternative coding techniques may provide better solutions.

For developers using compilers that do not support inner classes, try using the simulation techniques. You will find that they are very useful, especially for event handling chores.

References

Inner Classes Specification, Sun Microsystems, Inc., 1997.

Comments: Send your comments about this White Paper to [Josph McIntyre - joe@vnet.ibm.com](mailto:joe@vnet.ibm.com)

Java™ is a trademark of Sun Microsystems, Inc.

Microsoft, Windows and Windows NT are registered trademarks of Microsoft Corporation.

Other companies, products, and service names may be trademarks or service marks of others.

Copyright Trademark

[Java Feature Page](#) [Java Home](#)

