Inner Classes

Source	http://www.javaranch.com/campfire/StoryInner.jsp

One Object shares deeply personal feelings. Next... on a very, special, Campfire Story.

Being an object like me isn't as fun as you might think.

It gets lonely...

Out here...
on the heap...
alone.

Not to mention the horror, the emotional devastation when you feel your last reference slip away and it hits you -- you've just become food for the garbage collector.

But you know what helps? Having an inner class. An inner class can ease the loneliness... as long as somebody makes an *instance* of that inner class. All I really want is someone to bond with.

Someone to **share my most private thoughts** (and variables and methods). Someone who knows EVERYTHING about me. An intimate relationship shared between two objects -- an outer and an inner.

I'm very protective of my inner class. If somebody wants to instantiate my inner class, they MUST go through me-- an object of the outer class.

My inner class can't exist on its own. I, as an instance of the outer class, can live on my own (however unhappily). You don't have to make an instance of an inner class in order to have an instance of the outer class. But you can NEVER make an instance of my inner class without an outer object to "bind" it to.

My inner class needs me.

We have that special bond.

It makes life on the garbage-collectible heap bearable.

Here's how to do a little object matchmaking of your own:

```
class Outer
{
    private int size ;
    private String thoughts = "My outer thoughts";

    class Inner
    {
        String innerThoughts = "My inner thoughts";

        void doStuff()
        {
            // inner object has its own "this"
            System.out.println( innerThoughts );

            // and it also has a kind of "outer this"
            // even for private data of outer class
            System.out.println(thoughts);
        }
    }
}
```

OK, but nothing really happens until somebody makes an instance of BOTH classes...

```
class TestMe
{

    public static void main( String args[] )
    {
        // instantiate me, the outer object
        Outer o = new Outer();

        // Inner i = new Inner();
        // NO! Can't instantiate Inner by itself!
```

```
        Outer.Inner i = o.new Inner();
        // now I have my special inner object
        i.doStuff();
        // OK to call methods on inner object
    }

}
```

You can also instantiate both the outer class and inner class at the same time:

```
  Inner i = new Outer().new Inner();
```

I know that looks odd, but it shows that you need an outer object, so that you can ask it to make an inner object. In this example, you didn't even keep a reference to the outer object... only the inner object, "i". The inner object "i" still knows its outer object... its "outer this". (By the way, there is no keyword "outer this" -- that's just a concept for the way inner objects behave. They behave as if the outer object's variables were their own.)

I hate static!

You've probably heard about **static inner classes**. Well, they don't deserve to be called inner classes!

A static inner class (an inner class marked as static) looks like this:

```
class Outer
{
    static class Inner
    {
    }
}
```

I don't like them because they don't give me that special object-to-object bond. In fact, static inner classes aren't even supposed to be called inner classes at all. Technically, they are "**top-level nested classes**".

A static nested class can be instantiated, but the object created doesn't share any special relationship with an outer object.

The static nested class is tied only to the outer **class**, ***not an instance*** of the outer class.

```
     Outer.Inner i = new Outer.Inner();
```

That's why you can make an instance of the static nested class *without* having an instance of the outer class, just the way you can call static methods of a class without having any instances of that class. A top-level nested class is little more than another way to control namespace.

But let's go back to inner classes; they're so much more meaningful. And did you know that I can bond with an instance of my inner class even when I don't know the NAME of my inner class? For convenience, you can get an **instance** of an inner class and **make that inner class** at the **same time.**

It works like this...

Imagine you (the programmer) are making your nice GUI program and you decide that you need to know when the user clicks your GO button. "I reckon I need an ActionListener object", you say to yourself. So you type:

```
goButton.addActionListener([object goes here]);
```

And then you slap your forehead as you realize... "I can't make an **instance**... I don't even HAVE an ActionListener **class**!"

You never made yourself a class that implements the ActionListener interface.

Not a problem.

You can create a new class which implements the ActionListener interface, AND make an instance of that new class -- **all inside the parameter to the Button object's addActionListener() method.**
How cool is that?

It looks like this:

```
goButton.addActionListener
(
    new ActionListener()
    {
        public void actionPerformed( ActionEvent e )
        {
            doImportantStuff();
        }
    }
);
```

It works like this:

```
  new ActionListener()
```

says to the compiler: "Create an instance of a new, unnamed class which **implements the ActionListener interface**..."

And after that opening curly brace (shown above in green) you define the new unnamed class...

```
public void actionPerformed( actionEvent e )
{
    doImportantStuff();
}
```

That actionPerformed method is the same one you would be forced to define in any class which implements the ActionListener interface. But this new class has no name. That's why its called an **anonymous** inner class.

And notice that you did not say "new MyActionClass()". You said, "new ActionListener()". But you aren't making an instance of ActionListener, you're making an instance of your new anonymous class which implements the ActionListener interface.

"But wait!" you say, "What if I don't want to implement an interface... what if I want to make an anonymous inner class that **extends** another class?"

Once again, No Problem-o.

Whatever you say after the "new" as in "new Something()", if Something is an interface, then the anonymous class implements that interface (and must define all the methods of that interface). But if Something is a class, then your anonymous class automatically becomes a subclass of that class. This is perfect for the event adapter classes like WindowAdapter.

Finally, don't forget to close the parameter to the `goButton.addActionListener(`

by finishing off with a closing parentheses and the semicolon ending that statement...

```
);
```

Programmers always seem to forget that last little semicolon way down there, since the statement started waaaaaay up above somewhere.

Danger Danger!

Now I feel compelled to warn you about one thing with anonymous inner classes, or any other inner class that you define **inside** a method (as opposed to inside the class but NOT inside a method). **The inner class can't use local variables from the method in which the inner class is defined!**
After all, at the end of that method, the local variables will be blown away. Poof. Gone. History. That inner object you created from that inner class might still be alive and kickin' out on the heap

long after that local variable has gone out of scope.
You can, however, use local variables that are declared final, because the compiler takes care of that for you in advance. But that's it -- no method parameters from that method, and no local variables.

Another warning about all inner classes is that they can't declare any static members unless they are compile-time constants and are primitives or Strings. (This does not apply to static nested classes, of course). But don't worry -- the compiler will stop you if you try.

Finally, you should know that some programmers really dislike inner classes, and especially anonymous ones. Some folks claim they're not very object-oriented, and others get their feathers ruffled over "encapsulation violations" because the inner object can access the private data of the outer object.

Well you know what I say?
That's the point!

It's that special relationship, that intimate bond, that makes inner classes so practical. Otherwise you'd have to make constructors for your inner class, and pass references to variables, etc. when you instantiate the inner class... all the things you have to do with a plain old non-inner class, which you have to treat like an outsider.

But when you're trying to decide if inner classes are right for you, please... think of me...

the poor, lonely, object...

on the heap...

alone . 

~~~ End of Article ~~~