
PREVIOUS Chapter 5
Objects in Java NEXT



# 5.9 Inner Classes

We've left out something important in our discussion of Java classes so far: a large and relatively recent heap of syntactic sugar called *inner classes*. Simply put, classes in Java can be declared at any level of scope. That is, you can declare a class within any set of curly braces (that is, almost anywhere that you could put any other Java statement) and its visibility is limited to that scope in the same way that the name of a variable or method would be. Inner classes are a powerful and aesthetically pleasing facility for structuring code.[7] Their even sweeter cousins, *anonymous inner classes*, are another powerful shorthand that make it seem like you can create classes dynamically within Java's statically typed environment.

> [7] The implementation of Java's inner classes draws on experience from the language Beta, and other block structured languages such as Pascal, and Scheme.

However, if you delve into the inner workings of Java, inner classes are not quite as aesthetically pleasing or dynamic. We said that they are syntactic sugar; by this we mean that they let you leverage the compiler by writing a few lines of code that trigger a lot of behind-the-scenes work somewhere between the compiler's front end and the byte-code. Inner classes rely on code-generation; they are a feature of the Java language, but not of the Java virtual machine. As a programmer you may never need be aware of this; you can simply rely on inner classes like any other language construct. However, you should know a little about how inner classes work, to better understand the results and a few potential side effects.

To this point, all of our classes have been *top level* classes. We have declared them, free standing, at the package level. Inner classes are essentially nested classes, like this:

```
Class Animal {
    Class Brain {
        ...
    }
}
```

Here the class `Brain` is an inner class: it is a class declared inside the scope of class `Animal`. Although the details of what that means require a fair bit of explanation, we'll start by saying that the Java language tries to make the meaning, as much as possible, the same as for the other Java entities (methods and variables) living at that level of scope. For example, let's add a method to the `Animal` class:

```
Class Animal {
    Class Brain {
        ...
    }
    void performBehavior() { ... }
}
```

Both the inner class `Brain` and the method `performBehavior()` are within the scope of `Animal`. Therefore, anywhere within `Animal` we can refer to `Brain` and `performBehavior()` directly, by name. Within `Animal` we can call the constructor for `Brain` (`new Brain()`) to get a `Brain` object, or invoke `performBehavior()` to carry out that method's function. But neither `Brain` nor `performBehavior()` are accessible outside of the class `Animal` without some additional qualification.

Within the body of the `Brain` class and the body of the `performBehavior()` method, we have direct access to all of the other methods and variables of the `Animal` class. So, just as the `performBehavior()` method could work with the `Brain` class and create instances of `Brain`, code within the `Brain` class can invoke the `performBehavior()` method of `Animal` as well as work with any other methods and variables declared in `Animal`.

That last bit has important consequences. From within `Brain` we can invoke the method `performBehavior()`; that is--from within an instance of `Brain` we can invoke the `performBehavior()` method of an instance of `Animal`. Well, which instance of `Animal`? If we have several `Animal` objects around (say, a few `Cat`s and `Dog`s), we need to know whose `performBehavior()` method we are calling. What does it mean for a class definition to be "inside" another class definition? The answer is that a `Brain` object always lives within a single instance of `Animal`: the one that it was told about when it was created. We'll call the object that contains any instance of `Brain` its *enclosing instance*.

A `Brain` object cannot live outside of an enclosing instance of an `Animal` object. Anywhere you see an instance of `Brain`, it will be tethered to an instance of `Animal`. Although it is possible to construct a `Brain` object from elsewhere (i.e., another class), `Brain` always requires an enclosing instance of `Animal` to "hold" it. We'll also say now that if `Brain` is to be referred to from outside of `Animal` it acts something like an `Animal.Brain` class. And just as with the `performBehavior()` method, modifiers can be applied to restrict its visibility. There is even an interpretation of the `static` modifier, which we'll talk about a bit later. However, the details are somewhat boring and not immediately useful, so you should consult a full language reference for more info (like O'Reilly's *Java Language Reference*, Second Edition). So before we get too far afield, let's turn to a more compelling example.

A particularly important use of inner classes is to make *adapter classes*. An adapter class is a "helper" class that ties one class to another in a very specific way. Using adapter classes you can write your classes more naturally, without having to anticipate every conceivable user's needs in advance. Instead, you provide adapter classes that marry your class to a particular interface.

As an example, let's say that we have an `EmployeeList` object:

```
public class EmployeeList {
    private Employee [] employees = ... ;
    ...
}
```

`EmployeeList` holds information about a set of employees, representing some view of our database. Let's say that we would like to have `EmployeeList` provide its elements as an enumeration (see Chapter 7, *Basic Utility Classes*). An enumeration is a simple interface to a set of objects that looks like this:

```
// the java.util.Enumeration interface
public interface Enumeration {
    boolean hasMoreElements();
    Object nextElement();
}
```

It lets us iterate through its elements, asking for the next one and

testing to see if more remain. The enumeration is a good candidate for an adapter class because it is an interface that our `EmployeeList` can't readily implement itself. That's because an enumeration is a "one way", disposable view of our data. It isn't intended to be reset and used again, and therefore should be kept separate from the `EmployeeList` itself. This is crying out for a simple class to provide the enumeration capability. But what should that class look like?

Well, before we knew about inner classes, our only recourse would have been to make a new "top level" class. We would probably feel obliged to call it `EmployeeListEnumeration`:

```
class EmployeeListEnumeration implements Enumeration {
    // lots of knowledge about EmployeeList
    ...
}
```

Here we have a comment representing the machinery that the `EmployeeListEnumeration` requires. Think for just a second about what you'd have to do to implement that machinery. The resulting class would be completely coupled to the `EmployeeList`, and unusable in other situations. Worse, to function it must have access to the inner workings of `EmployeeList`. We would have to allow `EmployeeListEnumeration` access to the private array in `EmployeeList`, exposing this data more widely than it should be. This is less than ideal.

This sounds like a job for inner classes. We already said that `EmployeeListEnumeration` was useless without the `EmployeeList`; this sounds a lot like the "lives inside" relationship we described earlier. Furthermore, an inner class lets us avoid the encapsulation problem, because it can access all the members of its enclosing instance. Therefore, if we use an inner class to implement the enumeration, the array `employees` can remain `private`, invisible outside the `EmployeeList`. So let's just shove that helper class inside the scope of our `EmployeeList`:

```
public class EmployeeList {
    private Employee [] employees = ... ;

    // ...

    class Enumerator implements java.util.Enumeration {
        int element = 0;
        boolean hasMoreElements() {
            return  element < employees.length ;
        }
        Object nextElement() {
            if ( hasMoreElements() )
                return employees[ element++ ];
            else
                throw NoSuchElementException();
        }
    }

}
```

Now `EmployeeList` can provide an accessor method like the following to let other classes work with the list:

```
    ...
    Enumeration getEnumeration() {
        return new Enumerator();
    }
```

One effect of the move is that we are free to be a little more familiar in the naming of our enumeration class. Since it is no longer a top level class, we can give it a name that is only appropriate within the `EmployeeList`. In this case, we've named it `Enumerator` to emphasize what it does--but we don't need a name like `EmployeeEnumerator` that shows the relationship to the `EmployeeList` class, because that's implicit. We've also filled in the guts of the `Enumerator` class. As you can see, now that it is inside the scope of `EmployeeList`, `Enumerator` has direct

access to its private members, so it can directly access the `employees` array. This greatly simplifies the code and maintains the compile-time safety.

Before we move on, we should note that inner classes can have constructors, even though we didn't need one in this example. They are in all respects real classes.

## Inner Classes within methods

Inner classes may also be declared withing the body of a method. Returning to the `Animal` class, we could put `Brain` inside the `performBehavior()` method if we decided that the class was only useful inside of that method.

```
Class Animal {
    void performBehavior() {
        Class Brain {
            ...
        }
    }
}
```

In this situation, the rules governing what `Brain` can see are the same as in our earlier example. The body of `Brain` can see anything in the scope of `performBehavior()` and, of course, above it. This includes local variables of `performBehavior()`, and its arguments. This raises a few questions.

`performBehavior()` is a method, and methods have limited lifetimes. When they exit their local variables normally disappear into the stacky abyss. But an instance of `Brain` (like any object) lives on as long as it is referenced. So Java makes sure that any local variables used by instances of `Brain` created within an invocation of `performBehavior()` also live on. Furthermore, all of the instances of `Brain` that we make within a single invocation of `performBehavior()` will see the same local variables.

### Static Inner Classes

We mentioned earlier that the inner class `Brain` of the class `Animal` could in some ways be considered an `Animal.Brain` class. That is, it is possible to work with a `Brain` from outside the `Animal` class, using just such a qualified name: `Animal.Brain`. But given that our `Animal.Brain` class always requires an instance of an `Animal` as its enclosing instance, some explicit setup is needed.[8]

> [8] Specifically, we would have to follow a design pattern and pass a reference to the enclosing instance of `Animal` into the `Animal.Brain` constructor. See a language reference for more information. We don't expect you to run into this situation very often.

But there is another situation where we might use inner classes by name. An inner class that lives within the body of a top level class (not within a method or another inner class) can be declared `static`. For example:

```
class Animal  {
    static class MigrationPattern {
        ...
    }
    ...
}
```

A static inner class such as this acts just like a new top level class called `Animal.MigrationPattern`; we can use it without regard to any enclosing instances. Although this seems strange, it is not inconsistent since a static member never has an object instance

associated with it. The requirement that the inner class be defined directly inside a top level class ensures that an enclosing instance won't be needed. If we have permission, we can create an instance of the class using the qualified name:

```
Animal.MigrationPattern stlToSanFrancisco = new Animal.MigrationPattern();
```

As you see, the effect is that `Animal` acts something like a mini-package, holding the `MigrationPattern` class. We can use all of the standard visibility modifiers on inner classes, so a static inner class could be private, protected, default, or publicly visible.

**Anonymous Inner Classes**

Now we get to the best part.

As a general rule, the more deeply encapsulated and limited in scope our classes are, the more freedom we have in naming them. We saw this in our enumeration example. This is not just a purely aesthetic issue. Naming is an important part of writing readable and maintainable code. We generally want to give things the most concise and meaningful names possible. A corollary to this is that we prefer to avoid doling out names for purely ephemeral objects that are only going to be used once.

Anonymous inner classes are an extension of the syntax of the `new` operation. When you create an anonymous inner class, you combine the class's declaration with the allocation of an instance of that class (much like the way you can declare a variable of the type of an un-named structure in C). After the `new` operator, you specify either the name of a class or an interface, followed by a class body. The class body becomes an inner class, which either extends the specified class or, in the case of an interface, is expected to implement the specified interface. A single instance of the class is created and returned as the value.

For example, we could do away with the declaration of the `Enumerator` class in the `EmployeeList` example by using an anonymous inner class in the `getEnumeration()` method:

```
...

Enumeration getEnumeration() {

    return new Enumeration() {
        int element = 0;
        boolean hasMoreElements() {
            return  element < employees.length ;
        }
        Object nextElement() {
            if ( hasMoreElements() )
                return employees[ element++ ];
            else
                throw NoSuchElementException();
        }
    };
}
```

Here we have simply moved the guts of `Enumerator` into the body of an anonymous inner class. The call to `new` implicitly constructs the class and returns an instance of the class as its result. Note the extent of the curly braces, and the semi-colon at the end. It is a single statement.

But the code above certainly does not improve readability. Inner classes are best used when you want to implement a few lines of code, where the verbiage and conspicuousness of declaring a separate class detracts from the task at hand. Here's a better example: Suppose that we want to start a new thread to execute the

`performBehavior()` method of our `Animal`:

```
    new Thread ( new Runnable() {
        public void run() {  performBehavior();  }
    } ).start();
```

Here we have gone over to the terse side. We've allocated and started a new `Thread`, providing an anonymous inner class that implements the `Runnable` interface by calling our `performBehavior()` method. The effect is similar to using a method pointer in some other language; the inner class effectively substitutes the method we want called (`performBehavior()`) for the method the system wants to call (`run()`). However, the inner class allows the compiler to check type consistency, which would be difficult (if not impossible) with a true method pointer. At the same time, our anonymous adapter class with its three lines of code is much more efficient and readable than creating a new, top level adapter class named `AnimalBehaviorThreadAdapter`.

While we're getting a bit ahead of the story, anonymous adapter classes are a perfect fit for event handling (which we'll cover fully in Chapter 10, *Understand the Abstract Windowing Toolkit*). Skipping a lot of explanation, let's say you want the method `handleClicks()` to be called whenever the user clicks the mouse. You would write code like this:

```
addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent e) { handleClicks(e); }
});
```

In this case, the anonymous class extends the AWT's `MouseAdapter` class, by overriding its `mouseClicked()` method to call our method. A lot is going on in a very small space, but the result is clean, readable code. You get to assign method names that are meaningful to you, while allowing Java to do its job of type checking.

### `this` and scoping

Sometimes an inner class may want to get a handle on its "parent" enclosing instance. It might want to pass a reference to its parent, or to refer to one of the parent's variables or methods that has been hidden by one of its own. For example:

```
class Animal {
    int size;
    class Brain {
        int size;
    }
}
```

Here, as far as `Brain` is concerned, the variable `size` in `Animal` is hidden by its own version.

Normally an object refers to itself using the special `this` reference (implicitly or explicitly). But what is the meaning of `this` for an object with one or more enclosing instances? The answer is that an inner class has multiple `this` references. You can specify which `this` you want by prepending the name of the class. So, for instance (no pun intended), we can get a reference to our `Animal` from within `Brain` like so:

```
...
class Brain {
    Animal ourAnimal = Animal.this;
    ...
```

Similarly, we could refer to the `size` variable in `Animal`:

```
...
class Brain {
    int animalSize = Animal.this.size;
```

. . .

### How do inner classes really work?

Finally, we'll get our hands dirty and take a look at what's really going on when we use an inner class. We've said that the compiler is doing all of the things that we had hoped to forget about. Let's see what's actually happening. Try compiling our simple example:

```
class Animal {
    class Brain {
    }
}
```

(Oh, come on, do it...)

What you'll find is that the compiler generates two *.class* files:

Animal.class
Animal$Brain.class

The second file is the class file for our inner class. Yes, as we feared, inner classes are really just compiler magic. The compiler has created the inner class for us as a normal, top level class and named it by combining the class names with a dollar sign. The dollar sign is a valid character in class names, but is intended for use only by automated tools in this way. (Please don't start naming your classes with dollar signs). Had our class been more deeply nested, the intervening inner class names would have been attached in the same way to generate a unique top level name.

Now take a look at it using the *javap* utility:

```
# javap 'Animal$Brain'
class Animal$Brain extends java.lang.Object
{
    Animal$Brain(Animal);
}
```

You'll see that the compiler has given our inner class a constructor that takes a reference to an `Animal` as an argument. This is how the real inner class gets the handle on its enclosing instance.

The worst thing about these additional class files is that you need to know they are there. Utilities like *jar* don't automatically find them; when you are invoking a utility like *jar*, you need to specify these files explicitly, or use a wild card that finds them.

### Security Implications

Given what we just saw above--that the inner class really does exist as an automatically generated top level class--how does it get access to private variables? The answer, unfortunately, is that the compiler is forced to break the encapsulation of your object and insert accessor methods so that the inner class can reach them. The accessor methods will be given package level access, so your object is still safe within its package walls, but it is conceivable that this difference could be meaningful if people were allowed to create new classes within your package.

The visibility modifiers on inner classes also have some problems. Current implementations of the virtual machine do not implement the notion of a private or protected class within a package, so giving your inner class anything other than public or default visibility is only a compile-time guarantee.

It is difficult to conceive of how these security issues could be abused, but it is interesting to note that Java is straining a bit to stay

within its original design.

PREVIOUS
Interfaces

HOME
BOOK INDEX

NEXT
The Object and Class Classes

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA