NESTED CLASSES

A class seems like it should be a pretty important thing. A class is a high-level building block of a program, representing a potentially complex idea and its associated data and behaviors. I've always felt a bit silly writing tiny little classes that exist only to group a few scraps of data together. However, such trivial classes are often useful and even essential. Fortunately, in Java, I can ease the embarrassment, because one class can be nested inside another class. My trivial little class doesn't have to stand on its own. It becomes part of a larger more respectable class. This is particularly useful when you want to create a little class specifically to support the work of a larger class. And, more seriously, there are other good reasons for nesting the definition of one class inside another class.

In Java, a nested class is any class whose definition is inside the definition of another class. Nested classes can be either named or anonymous. I will come back to the topic of anonymous classes later in this section. A named nested class, like most other things that occur in classes, can be either static or non-static.

The definition of a static nested class looks just like the definition of any other class, except that it is nested inside another class and it has the modifier `static` as part of its declaration. A static nested class is part of the static structure of the containing class. It can be used inside that class to create objects in the usual way. If it has not been declared private, then it can also be used outside the containing class, but when it is used outside the class, its name must indicate its membership in the containing class. This is similar to other static components of a class: A static nested class is part of the class itself in the same way that static member variables are parts of the class itself.

For example, suppose a class named *WireFrameModel* represents a set of lines in three-dimensional space. (Such models are used to represent three-dimensional objects in graphics programs.) Suppose that the *WireFrameModel* class contains a static nested class, *Line*, that represents a single line. Then, outside of the class *WireFrameModel*, the *Line* class would be referred to as `WireFrameModel.Line`. Of course, this just follows the normal naming convention for static members of a class. The definition of the *WireFrameModel* class with its nested *Line* class would look, in outline, like this:

```
public class WireFrameModel {

   . . . // other members of the WireFrameModel class

   static public class Line {
         // Represents a line from the point (x1,y1,z1)
         // to the point (x2,y2,z2) in 3-dimensional space.
      double x1, y1, z1;
      double x2, y2, z2;
   } // end class Line

   . . . // other members of the WireFrameModel class

} // end WireFrameModel
```

Inside the *WireFrameModel* class, a *Line* object would be created with the constructor "`new Line()`". Outside the class, "`new WireFrameModel.Line()`" would be used.

A static nested class has full access to the static members of the containing class, even to the private members. Similarly, the containing class has full access to the members of the nested class. This can be another motivation for declaring a nested class, since it lets you give one class access to the private members of another class without making those members generally available to other classes. Note also that a nested

class can itself be private, meaning that it can only be used inside the class in which it is nested.

When you compile the above class definition, two class files will be created. Even though the definition of *Line* is nested inside *WireFrameModel*, the compiled *Line* class is stored in a separate file. The name of the class file for *Line* will be `WireFrameModel$Line.class`.

Non-static nested classes are referred to as inner classes. Inner classes are not, in practice, very different from static nested classes, but a non-static nested class is actually associated with an object rather than to the class in which it is nested. This can take some getting used to.

Any non-static member of a class is not really part of the class itself (although its source code is contained in the class definition). This is true for inner classes, just as it is for any other non-static part of a class. The non-static members of a class specify what will be contained in objects that are created from that class. The same is true -- at least logically -- for inner classes. It's as if each object that belongs to the containing class has its **own copy** of the nested class. This copy has access to all the instance methods and instance variables of the object, even to those that are declared `private`. The two copies of the inner class in two different objects differ because the instance variables and methods they refer to are in different objects. In fact, the rule for deciding whether a nested class should be static or non-static is simple: If the nested class needs to use any instance variable or instance method from the containing class, make the nested class non-static. Otherwise, it might as well be static.

From outside the containing class, a non-static nested class has to be referred to using a name of the form **variableName**.**NestedClassName**, where **variableName** is a variable that refers to the object that contains the class. This is actually rather rare, however. A non-static nested class is generally used only inside the class in which it is nested, and there it can be referred to by its simple name.

In order to create an object that belongs to an inner class, you must first have an object that belongs to the containing class. (When working inside the class, the object "`this`" is used implicitly.) The inner class object is permanently associated with the containing class object, and it has complete access to the members of the containing class object. Looking at an example will help, and will hopefully convince you that inner classes are really very natural. Consider a class that represents poker games. This class might include a nested class to represent the players of the game. This structure of the *PokerGame* class could be:

```
public class PokerGame {  // Represents a game of poker.

    class Player {   // Represents one of the players in this
game.
        .
        .
        .
    } // end class Player

    private Deck deck;       // A deck of cards for playing the
game.
    private int pot;          // The amount of money that has
been bet.

    .
    .
    .
```

```
} // end class PokerGame
```

If `game` is a variable of type *PokerGame*, then, conceptually, `game` contains its own copy of the *Player* class. In an instance method of a *PokerGame* object, a new *Player* object would be created by saying "`new Player()`", just as for any other class. (A *Player* object could be created outside the *PokerGame* class with an expression such as "`game.new Player()`". Again, however, this is rare.) The *Player*object will have access to the `deck` and `pot` instance variables in the *PokerGame* object. Each `PokerGame` object has its own `deck` and `pot` and `Players`. Players of that poker game use the deck and pot for that game; players of another poker game use the other game's deck and pot. That's the effect of making the *Player* class non-static. This is the most natural way for players to behave. A *Player* object represents a player of one particular poker game. If *Player* were a **static** nested class, on the other hand, it would represent the general idea of a poker player, independent of a particular poker game.

Source : http://math.hws.edu/javanotes/c5/s7.html