Introduction to Illustrated C# 7

I took another look at this eBook and realized that the introductory material isn't really necessary at first. Yeah, it gives a certain amount of perhaps interesting background, but it really requires a somewhat experienced knowledge of Microsoft technologies. For example, if you've never heard of COM, then you don't really need to understand why the .NET Framework is better, and what the interaction is between C# and COM.

I suggest starting at page 23 (Overview of C# Programming). Read the first few pages about the canonical "Hello world" program, but here's my explanation.

First of all, I almost always like to explain something as this: Someone had a problem, and this (the thing I want to explain) was what they came up with to solve it. So this next part assumes you're working with the sample code on page 24. I'll reproduce it here:

```
using System;
namespace Simple {
   class Program {
      static void Main() {
         Console.WriteLine("Hi there!");
      }
   }
}
```

Problem: The C# library (technically a *class* library, but we'll worry about that later) is extremely large, and you wouldn't want to have to load in ***all*** the library into memory, just to write "Hi there!". So you must somehow specify what libraries you need. Hence the "using" statement. So in our simple "Hello world" example, we'll tell the C# compiler that we want to use the System library. The syntax is "using System;". How do you know to do this? And why do you use "System" instead of, say, "Sys" (or even “sys”)? Well, you can't start from absolutely no knowledge at all. And what other things can follow "using "? Answer: You don't know. That's why you need to read introductory books or have a human teacher (in this case, moi).

Note that C#'s *using* statement ends with a ; and is comparable to Python's *import* statement.

Problem: As we'll see shortly, we can define our own *class*es (more on this soon, I promise). In this example it's called *Program*. But suppose some library you use also defines a class named *Program*. Well, that's easy enough to fix. Just change the name

Program to something else (e.g. *ColinProgram* -- unless *that* name is also in use somewhere!). But suppose you're using two different third-party libraries, from two different vendors. Both may define a class called *Program*. Or worse, suppose both define a class named, say, *Interest*, one referring to bank interest, and the other to things you might be interested in (e.g. hobbies). You have no control over those third-party libraries. So how do you distinguish them? The answer is that we have to qualify the class names involved. Surrounding class definitions with a *namespace* statement accomplishes this. So, if necessary, when referring to the *Program* class in your program, you could write *Simple.Program* (i.e. namespace name dot class name). In the case of third party libraries, it would be extremely inconsiderate (bordering on incompetence) to not wrap their class names in one (or more) namespaces. One convention that has been suggested is to use the base name of their web site, backwards. For example, *namespace com.MyCompanyName { ... }* so now you could refer to *com.MyCompany.Interest* and *com.OtherCompany.Interest* and there'd be no confusion.

So that's what's going on here. But with all that said, unless you're a library writer, you don't actually need this extra level of qualification, and you can usually leave the *namespace* command out entirely. But it's a simple thing to add and doesn't hurt. So I'd class (no pun intended - for a change) this as "unnecessary but comes under the heading of Best Practices".

Problem: Assume you're writing a large-ish program, thousands of lines long. Would you place the entire 5,000 (say) lines all in one source module? You could, but hopefully it's obvious that you'd break it up into smaller source files, each representing a specific subset of your program logic. One module might contain all the database routines. Another might let you work with your graphics routines. And so on. And even within an individual source file, you wouldn't have a monolithic single function; you'd break down your logic into subroutines. Well, classes are a more disciplined way of doing things. Many years of experience has taught up that a good way of doing this is to take a data structure and associate it with a set of functions (called generically *methods*) that work with the data structure.

Now our class, *Program* has no data fields, and consists entirely of a single function, so it isn't a great example of this. But we could create a simple class called, say, *Person*, with fields for a first name, a last name, their date of birth, an address, and so forth. There wouldn't necessarily be much to do with such a data structure (other than to create it in the first place), but you could do things like define functions (methods!) that format the first and last names into a single output string (call it FullName), or calculate the person's age from their date of birth and today's date). See chapter 8 in the book for more examples of this.

Problem: We've got one or more source modules in our program, each with one or more classes, each with one or more methods. When the program starts, which one is executed first? Answer: it looks for a *static* method (don't sweat the *static* keyword at this point; it's necessary, but not otherwise important) called "Main". In this example, it's declared as *void*, which means it doesn't return anything to its caller. It's possible to return an integer (*void int Main()*) which is often used to tell whoever invoked the program (perhaps a shell script) something about how things went inside the execution of the program. By convention, returning zero indicates that the program worked fine. Other values might indicate things like: no parameters supplied, but you were expecting some; some error checking failed, such as an input filename referring to a non-existent file; an unexpected error, such as the program trying to divide by zero, and so on. But in this sample program, we don't return anything.

Problem: We want to write our “Hi there!” message to the user. Within the *System* namespace there is a class called *Console*. It has a method called *WriteLine* that takes (among other things) a simple string and displays it on the screen, with a trailing newline.

So that’s it! Yeah, I agree that this is a lot more complex than Python’s one-line “print(‘Hi there!’)”.

But as I emailed earlier, Guido von Rossum, the inventor of Python, has come around to realize that for other than simple programs, a more sophisticated approach to programming is useful, and a language that forces a certain discipline on you is actually a good thing. See (https://www.techrepublic.com/article/the-creator-of-python-on-how-the-programming-language-is-learning-from-typescript/).

And finally, adding a couple of *using* statements, an optional *namespace*, and a *class* statement, yeah, percentage-wise that’s a lot of overhead for such a simple program. But as soon as you start putting real code in there to actually accomplish something non-trivial, this overhead becomes negligible.