Intermediate Language (IL)—which is also known as Common Intermediate Language (CIL) and as Microsoft Intermediate Language (MSIL)—lies at the core of .NET. Whatever language you use to write your source code in, it will, if it's to run under the auspices of the .NET Framework, end up as IL. So if you want to understand at an advanced level how .NET works, then knowing a bit about IL is a huge advantage. If you understand some IL, you'll be able to do the following:

- Understand better how the managed code you write works under the hood. You'll have two sources of information: you can read the documentation, or you can examine the IL code generated by your compiler—which may give you insights that aren't covered by the documentation.

- Examine the code that the compiler for your usual language (such as C++, C#, or Visual Basic [VB]) emits, which sometimes can help you in debugging or in designing code that's optimized for performance. It may also help you understand some of the finer points of how your chosen language works.

- Actually write some code directly in IL—it's probably not that often that you'll want to do this, but you may occasionally find it useful to be able to take advantage of IL features not implemented in your normal language, in much the same way that in pre-.NET days people writing high-performance C or C++ code would occasionally drop to native assembly language to improve the performance of some particular function.

Obviously, if you're writing developer tools such as compilers or debuggers, understanding IL is a prerequisite!

Because of the importance of IL, I've decided to introduce the language before I do anything else in this book—which I'll do in this and the next chapter. In this chapter, I'll concentrate on the basics. You'll learn basic IL assembly syntax; how the abstract stack machine, on which IL is based, works; and how to code procedural flow control in IL. Finally, you'll look at IL errors and how to debug IL source code. Then in Chapter 2, I'll build on all this by showing how to code classes and structs and invoke instance methods. In that chapter I'll also cover some more advanced topics such as working with delegates and exceptions and calling into unmanaged code.

I won't go into every nuance of the language, and I won't cover some of the more advanced, and rarely used, features. This is, after all, a book about advanced .NET programming in general,
not a book about IL programming. I’m not expecting you to start writing all your code in IL—for most purposes that would be a pretty silly thing to do, and the only significant result would be to multiply your development and debugging time considerably. Rather, I’m working on the basis that understanding IL will help you get the most out of the .NET Framework in your high-level language development. So, while I’ll inevitably have to cover IL syntax, the emphasis in these chapters is on teaching you the basic concepts sufficiently so that you can read IL. Another motivation is that high-level managed languages do sometimes hide or give a misleading impression of how the .NET Framework implements certain tasks. C++, C#, and VB are all guilty of this to some extent. High-level languages tend to hide such implementation details in order to make things easy for you, but IL, the language that the Just-in-Time (JIT) compiler has to deal with, can hide nothing—so by learning IL, you can get the true picture of what is happening.

You’ll find this chapter starts at a fairly gentle pace in terms of IL concepts. I’m not assuming you’ve had any experience of programming in IL (or native machine code for that matter). I do, however, assume you’re experienced in your high-level language and understand object-oriented programming and the basic principles of the .NET Framework. At the end of these two chapters, you should have a good enough grasp of IL to be able to read most of the IL code generated by your compiler. In addition, this book’s appendix gives a comprehensive list of the meanings of every IL instruction, along with the corresponding opcodes, which you can refer to if you encounter any IL instructions not mentioned in this chapter.

I should also remind you that you don’t strictly have to read these IL chapters—most of the code samples throughout the rest of the book are in C#, so as long as you can read C# you’ll be able to get through most of the rest of the book. If you really feel daunted at the prospect of reading assembly code, feel free to skip ahead to Chapter 3. But I do think that if you have a sound grasp of the principles of IL, then your advanced .NET programming will benefit.

Introducing IL Assembly

IL itself has a binary format. Just as with native assembly language, an IL instruction is actually stored in its containing assembly as a binary number (an opcode), which means that it would be pretty pointless to use a text editor to read a file containing IL. However, just as with native executable code, an assembly language has been defined for IL that consists of textual mnemonic codes to represent the IL commands. This language is known as IL assembly—though since this is a long name, you’ll often hear it conveniently, albeit not strictly accurately, referred to just as IL, or sometimes as ILAsm or as IL source code. For example, the IL instruction to add two numbers together is the opcode 0xS8 (88 in decimal—in this book I follow the usual practice of prefixing hexadecimal numbers with 0x), but this instruction is represented in IL assembly by the string add. For obvious reasons, I’ll use assembly rather than the native IL code in this book. In effect, I’ll be teaching you IL assembly rather than straight IL. However, because of the obvious one-to-one correspondence between the IL assembly instructions and the IL instructions, this means for all practical purposes you’ll be learning IL as well. In this chapter, don’t worry too much about the actual format of how the opcodes are represented in assemblies—I’ll deal with that issue in Chapter 4, where you’ll examine assembly format in more detail. I’ll mention, though, that keeping file size small was one of the main design priorities for the binary format. Hence, most opcodes occupy just 1 byte, although some more rarely used opcodes occupy 2 bytes. Quite a few of the instructions also take arguments—numbers occupying anything from 1 to 4 bytes that follow the instruction in the assembly and provide more information about that instruction. For example, the call instruction, which invokes a method,