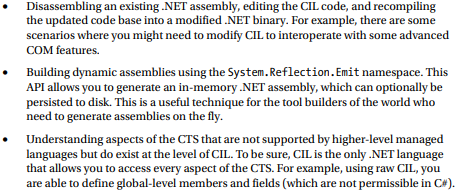
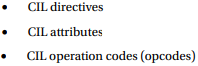
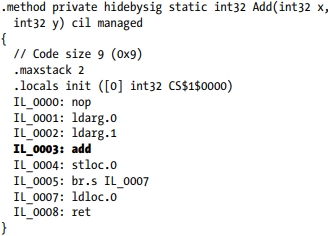
**Understanding CIL and the Role of Dynamic Assemblies** When you are building a full-scale .NET application, you will most certainly use C# (or a similar managed language such as Visual Basic), given its inherent productivity and ease of use. However, as you learned in the first chapter, the role of a managed compiler is to translate \*.cs code files into terms of CIL code, type metadata, and an assembly manifest. As it turns out, CIL is a full-fledged .NET programming language, with its own syntax, semantics, and compiler (ilasm.exe).

**Motivations for Learning the Grammar of CIL** CIL is the true mother tongue of the .NET platform. When you build a .NET assembly using your managed language of choice (C#, VB, F#, etc.), the associated compiler translates your source code into terms of CIL. Like any programming language, CIL provides numerous structural and implementation-centric tokens. Given that CIL is just another .NET programming language, it should come as no surprise that it is possible to build your .NET assemblies directly using CIL and the CIL compiler (ilasm.exe) that ships with the .NET Framework SDK.

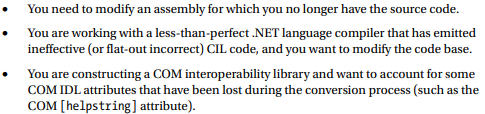
**Examining CIL Directives, Attributes, and Opcodes** When you begin to investigate low-level languages such as CIL, you are guaranteed to find new (and often intimidating-sounding) names for familiar concepts. For example, at this point in the text, if you were shown the following set of items **{new, public, this, base, get, set, explicit, unsafe, enum, operator, partial}**

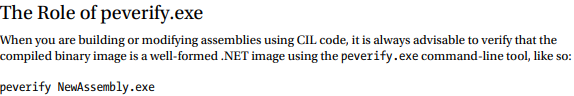
**The Role of CIL Directives** First up, there is a set of well-known CIL tokens that are used to describe the overall structure of a .NET assembly. These tokens are called directives. CIL directives are used to inform the CIL compiler how to define the namespaces(s), type(s), and member(s) that will populate an assembly.

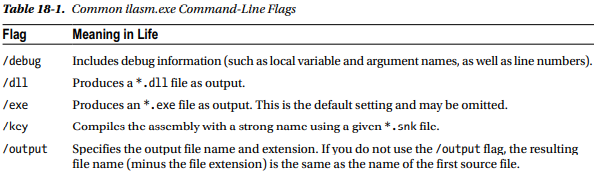
**The Role of CIL Attributes** In many cases, CIL directives in and of themselves are not descriptive enough to fully express the definition of a given .NET type or type member. Given this fact, many CIL directives can be further specified with various CIL attributes to qualify how a directive should be processed. For example, the .class directive can be adorned with the public attribute (to establish the type visibility), the extends attribute (to explicitly specify the type’s base class), and the implements attribute (to list the set of interfaces supported by the type).

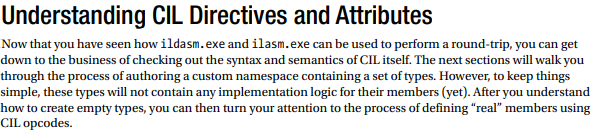
**The Role of CIL Opcodes** Once a .NET assembly, namespace, and type set have been defined in terms of CIL using various directives and related attributes, the final remaining task is to provide the type’s implementation logic. This is a job for operation codes, or simply opcodes. In the tradition of other low-level languages, many CIL opcodes tend to be cryptic and completely unpronounceable by us mere humans. For example, if you need to load a string variable into memory, you don’t use a friendly opcode named LoadString but rather ldstr.

**Pushing and Popping: The Stack-Based Nature of CIL** Higher-level .NET languages (such as C#) attempt to hide low-level CIL grunge from view as much as possible. One aspect of .NET development that is particularly well hidden is that CIL is a stack- based programming language. Recall from the examination of the collection namespaces (see Chapter 9) that the Stack class can be used to push a value onto a stack as well as pop the topmost value off of the stack for use. Of course, CIL developers do not use an object of type Stack to load and unload the values to be evaluated; however, the same pushing and popping mind-set still applies.

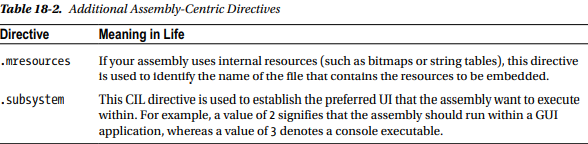
**Understanding Round-Trip Engineering** You are aware of how to use ildasm.exe to view the CIL code generated by the C# compiler (see Chapter 1). What you might not know, however, is that ildasm.exe allows you to dump the CIL contained within an assembly loaded into ildasm.exe to an external file. Once you have the CIL code at your disposal, you are free to edit and recompile the code base using the CIL compiler, ilasm.exe.

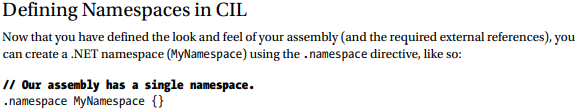


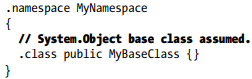


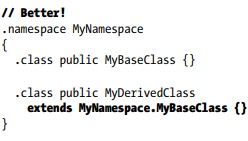


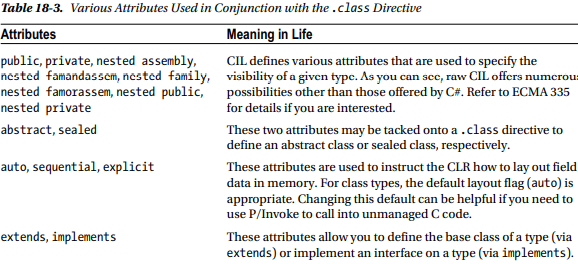
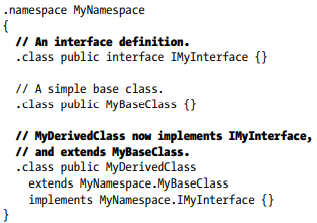
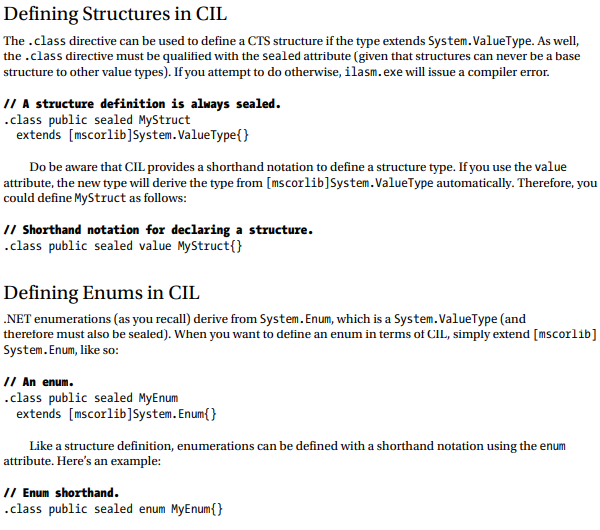


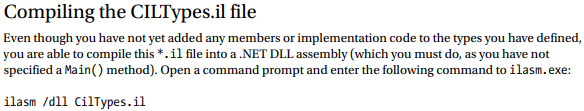
**Defining the Current Assembly in CIL** The next order of business is to define the assembly you are interested in building using the .assembly directive.

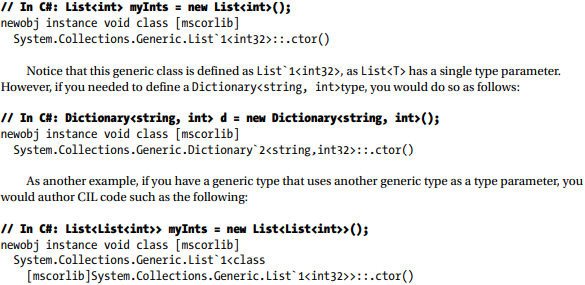




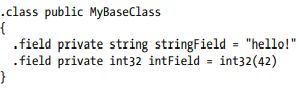
**Defining Class Types** in CIL Empty namespaces are not very interesting, so let’s now check out the process of defining a class type using CIL. Not surprisingly, the .class directive is used to define a new class. However, this simple directive can be adorned with numerous additional attributes, to further qualify the nature of the type. To illustrate, add a public class to your namespace named MyBaseClass. As in C#, if you do not specify an explicit base class, your type will automatically be derived from System.Object.

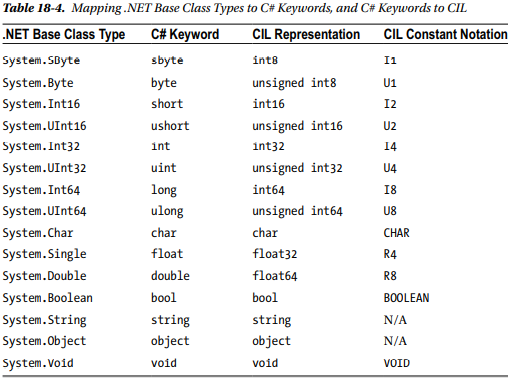
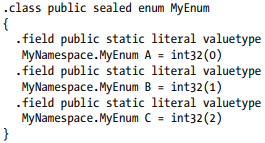
 **Defining and Implementing Interfaces in CIL** As odd as it might seem, interface types are defined in CIL using the .class directive. However, when the .class directive is adorned with the interface attribute, the type is realized as a CTS interface type.

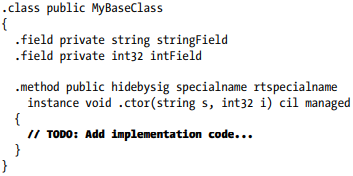
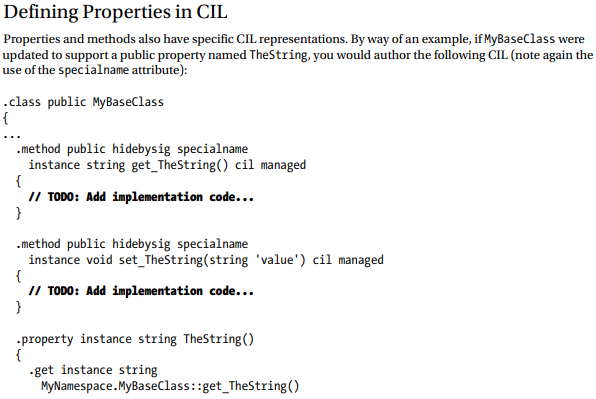
 **Defining Generics in CIL** Generic types also have a specific representation in the syntax of CIL. Recall from Chapter 9 that a given generic type or generic member may have one or more type parameters. For example, the List type has a single type parameter, while Dictionary has two. In terms of CIL, the number of type parameters is specified using a backward-leaning single tick (`), followed by a numerical value representing the number of type parameters. Like C#, the actual value of the type parameters is encased within angled brackets.



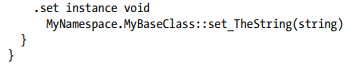
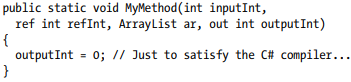
**.NET Base Class Library, C#, and CIL Data Type Mappings** Table 18-4 illustrates how a .NET base class type maps to the corresponding C# keyword and how each C# keyword maps into raw CIL. As well, Table 18-4 documents the shorthand constant notations used for each CIL type. As you will see in just a moment, these constants are often referenced by numerous CIL opcodes.

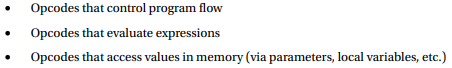


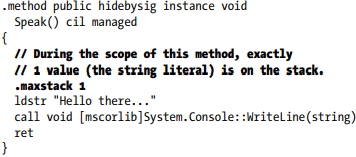
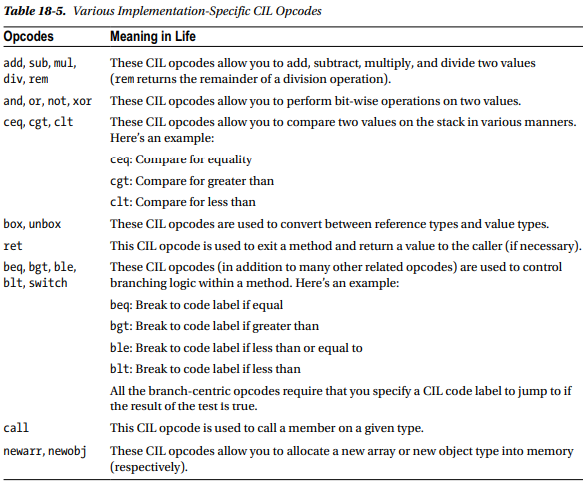


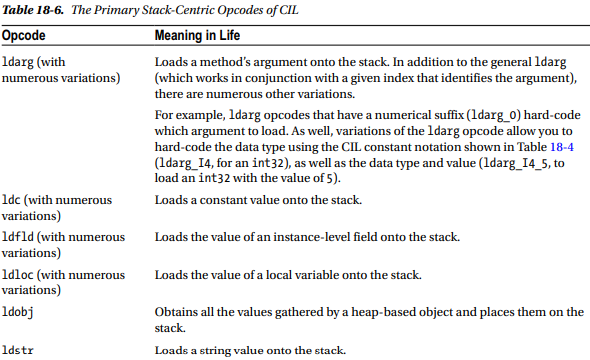
**Defining Type Constructors in CIL** The CTS supports both instance-level and class-level (static) constructors. In terms of CIL, instance-level constructors are represented using the .ctor token, while a static-level constructor is expressed via .cctor (class constructor). Both of these CIL tokens must be qualified using the rtspecialname (return type special name) and specialname attributes. Simply put, these attributes are used to identify a specific CIL token that can be treated in unique ways by a given .NET language. For example, in C#, constructors do not define a return type; however, in terms of CIL, the return value of a constructor is indeed void.

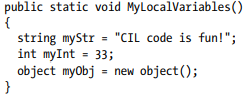
**Defining Member Parameters** In a nutshell, specifying arguments in CIL is (more or less) identical to doing so in C#. For example, each argument is defined by specifying its data type, followed by the parameter name. Furthermore, like C#, CIL provides a way to define input, output, and pass-by-reference parameters. As well, CIL allows you to define a parameter array argument (aka the C# params keyword), as well as optional parameters.

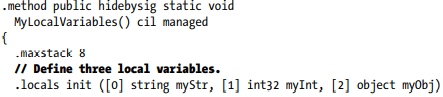


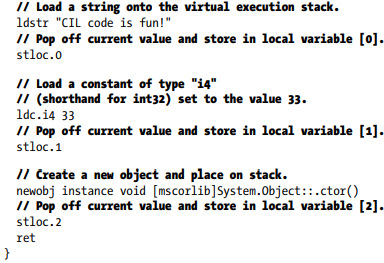
**Examining CIL Opcodes** The final aspect of CIL code you’ll examine in this chapter has to do with the role of various operational codes (opcodes). Recall that an opcode is simply a CIL token used to build the implementation logic for a given member.

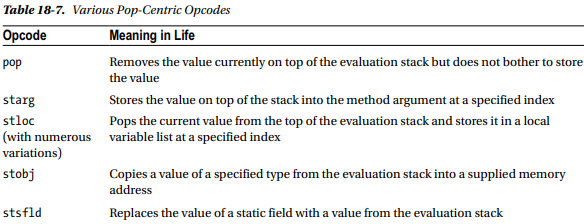
**The .maxstack Directive** When you write method implementations using raw CIL, you need to be mindful of a special directive named .maxstack. As its name suggests, .maxstack establishes the maximum number of variables that may be pushed onto the stack at any given time during the execution of the method. 

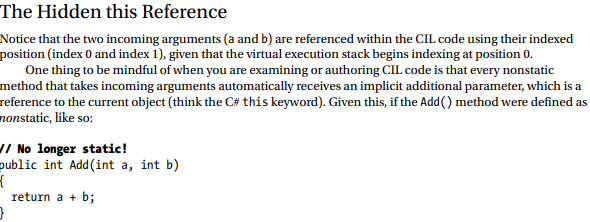
**Declaring Local Variables in CIL** Let’s first check out how to declare a local variable. Assume you want to build a method in CIL named MyLocalVariables() that takes no arguments and returns void. Within the method, you want to define three local variables of type System.String, System.Int32, and System.Object.

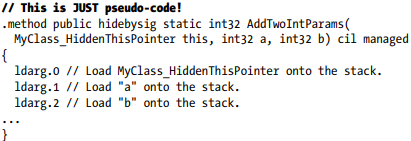
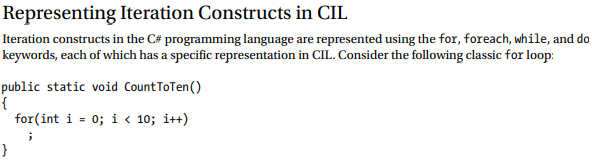


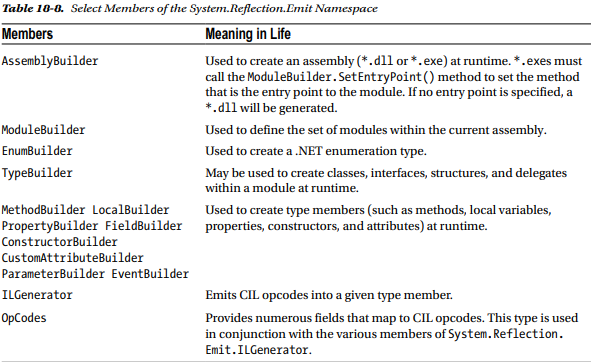
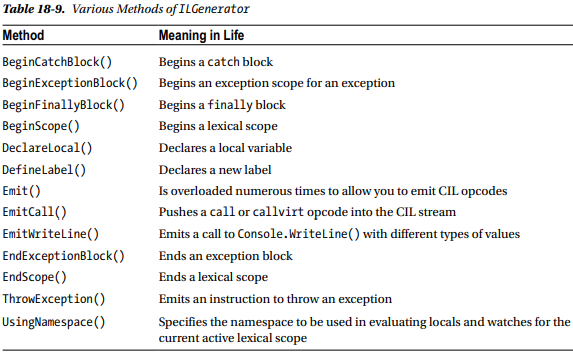


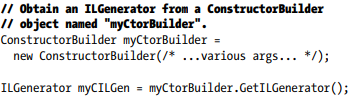


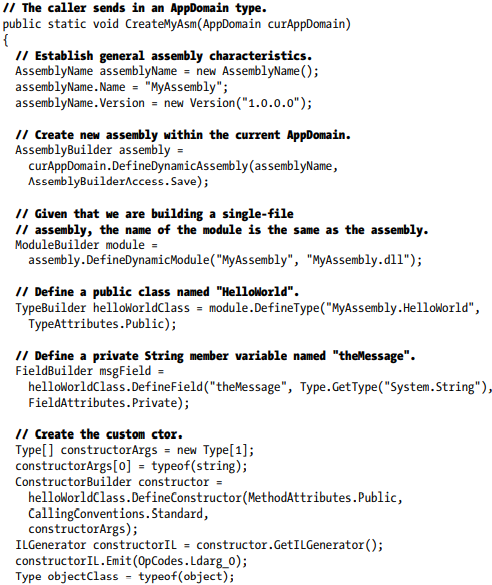
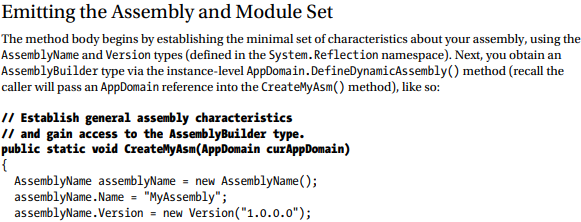
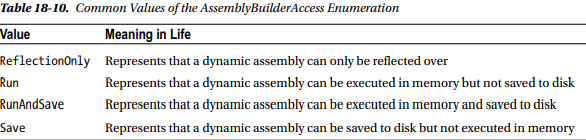
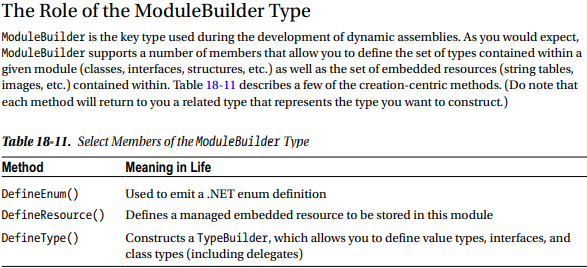


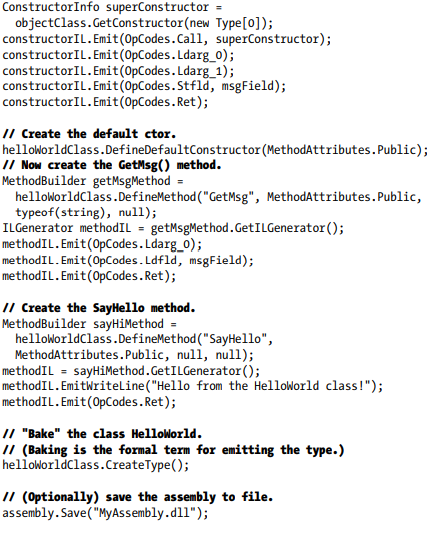


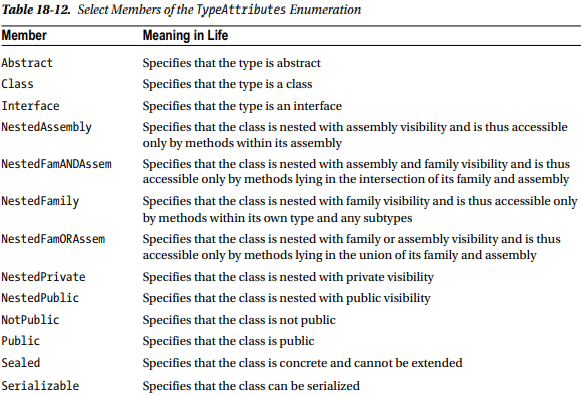


**Understanding Dynamic Assemblies** To be sure, the process of building a complex .NET application in CIL would be quite the labor of love. On the one hand, CIL is an extremely expressive programming language that allows you to interact with all the programming constructs allowed by the CTS. On the other hand, authoring raw CIL is tedious, errorprone, and painful. While it is true that knowledge is power, you might indeed wonder just how important it is to commit the laws of CIL syntax to memory. The answer is, “It depends.” To be sure, most of your .NET programming endeavors will not require you to view, edit, or author CIL code. However, with the CIL primer behind you, you are now ready to investigate the world of dynamic assemblies (as opposed to static assemblies) and the role of the System.Reflection.Emit namespace. The first question you may have is, “What exactly is the difference between static and dynamic assemblies?” By definition, static assemblies are .NET binaries loaded directly from disk storage, meaning they are located somewhere on your hard drive in a physical file (or possibly a set of files in the case of a multifile assembly) at the time the CLR requests them. As you might guess, every time you compile your C# source code, you end up with a static assembly. A dynamic assembly, on the other hand, is created in memory, on the fly, using the types provided by the System.Reflection.Emit namespace. The System.Reflection.Emit namespace makes it possible to create an assembly and its modules, type definitions, and CIL implementation logic at runtime. After you have done so, you are then free to save your in-memory binary to disk. This, of course, results in a new static assembly. To be sure, the process of building a dynamic assembly using the System.Reflection.Emit namespace does require some level of understanding regarding the nature of CIL opcodes

**The Role of the System.Reflection.Emit.ILGenerator** As its name implies, the ILGenerator type’s role is to inject CIL opcodes into a given type member. However, you cannot directly create ILGenerator objects, as this type has no public constructors; rather, you receive an ILGenerator type by calling specific methods of the builder-centric types (such as the MethodBuilder and ConstructorBuilder types). H







**Summary** This chapter provided an overview of the syntax and semantics of CIL. Unlike higher-level managed languages such as C#, CIL does not simply define a set of keywords but provides directives (used to define the structure of an assembly and its types), attributes (which further qualify a given directive), and opcodes (which are used to implement type members). You were introduced to a few CIL-centric programming tools and learned how to alter the contents of a .NET assembly with new CIL instructions using round-trip engineering. After this point, you spent time learning how to establish the current (and referenced) assembly, namespaces, types, and members. I wrapped up with a simple example of building a .NET code library and executable using little more than CIL, command-line tools, and a bit of elbow grease. Finally, you took an introductory look at the process of creating a dynamic assembly. Using the System.Reflection.Emit namespace, it is possible to define a .NET assembly in memory at runtime. As you have seen firsthand, using this particular API requires you to know the semantics of CIL code in some detail. While the need to build dynamic assemblies is certainly not a common task for most .NET applications, it can be useful for those of you who need to build support tools and other programming utilities