Expert .NET 2.0 IL Assembler

Serge Lidin

Apress®

Expert .NET 2.0 IL Assembler

ISBN-13: 978-1-59059-646-3

ISBN-10: 1-59059-646-3

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Ewan Buckingham
Technical Reviewers: Jim Hogg, Vance Morrison
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Sofia Marchant
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett
Assistant Production Director: Kari Brooks-Copony
Senior Production Editor: Laura Cheu
Compositor: Diana Van Winkle, Van Winkle Design
Proofreader: Linda Seifert
Indexer: Broccoli Information Management
Artist: Diana Van Winkle, Van Winkle Design
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 1

Simple Sample

This chapter offers a general overview of ILAsm, the MSIL assembly language. (MSIL stands for *Microsoft intermediate language*, which will soon be discussed in this chapter.) The chapter reviews a relatively simple program written in ILAsm, and then I suggest some modifications that illustrate how you can express the concepts and elements of Microsoft .NET programming in this language.

This chapter does not teach you how to write programs in ILAsm. But it should help you understand what the IL assembler (ILASM) and the IL disassembler (ILDASM) do and how to use that understanding to analyze the internal structure of a .NET-based program with the help of these ubiquitous tools. You'll also learn some intriguing facts about the mysterious affairs that take place behind the scenes within the common language runtime—intriguing enough, I hope, to prompt you to read the rest of the book.

Note For your sake and mine, I'll abbreviate *IL assembly language* as ILAsm throughout this book. Don't confuse it with ILASM, which is the abbreviation for the IL assembler (in other words, the ILAsm compiler) in the .NET documentation.

Basics of the Common Language Runtime

The .NET common language runtime is but one of many aspects of .NET, but it's the core of .NET. (Note that, for variety's sake, I'll sometimes refer to the common language runtime as *the runtime*.) Rather than focusing on an overall description of the .NET platform, I'll concentrate on the part of .NET where the action really happens: the common language runtime.

Note For excellent discussions of the general structure of .NET and its components, see *Introducing Microsoft .NET*, Third Edition (Microsoft Press, 2003), by David S. Platt, and *Inside C#*, Second Edition (Microsoft Press, 2002), by Tom Archer and Andrew Whitechapel.

Simply put, the common language runtime is a run-time environment in which .NET applications run. It provides an operating layer between the .NET applications and the underlying operating system. In principle, the common language runtime is similar to the runtimes of interpreted languages such as GBasic. But this similarity is only in principle: the common language runtime is not an interpreter.

The .NET applications generated by .NET-oriented compilers (such as Microsoft Visual C#, Microsoft Visual Basic .NET, ILAsm, and many others) are represented in an abstract, intermediate form, independent of the original programming language and of the target machine and its operating system. Because they are represented in this abstract form, .NET applications written in different languages can interoperate closely, not only on the level of calling each other's functions but also on the level of class inheritance.

Of course, given the differences in programming languages, a set of rules must be established for the applications to allow them to get along with their neighbors nicely. For example, if you write an application in Visual C# and name three items `MYITEM`, `MyItem`, and `myitem`, Visual Basic .NET, which is case insensitive, will have a hard time differentiating them. Likewise, if you write an application in ILAsm and define a global method, Visual C# will be unable to call the method because it has no concept of global (out-of-class) items.

The set of rules guaranteeing the interoperability of .NET applications is known as the Common Language Specification (CLS), outlined in Partition I of the Common Language Infrastructure standard of Ecma International and the International Organization for Standardization (ISO). It limits the naming conventions, the data types, the function types, and certain other elements, forming a common denominator for different languages. It is important to remember, however, that the CLS is merely a recommendation and has no bearing whatsoever on common language runtime functionality. If your application is not CLS compliant, it might be valid in terms of the common language runtime, but you have no guarantee that it will be able to interoperate with other applications on all levels.

The abstract intermediate representation of the .NET applications, intended for the common language runtime environment, includes two main components: metadata and managed code. *Metadata* is a system of descriptors of all structural items of the application—classes, their members and attributes, global items, and so on—and their relationships. This chapter provides some examples of metadata, and later chapters describe all the metadata structures.

The *managed code* represents the functionality of the application's methods (functions) encoded in an abstract binary form known as *Microsoft intermediate language* (MSIL) or *common intermediate language* (CIL). To simplify things, I'll refer to this encoding simply as *intermediate language* (IL). Of course, other intermediate languages exist in the world, but as far as our endeavors are concerned, let's agree that IL means MSIL, unless specified otherwise.

The runtime "manages" the IL code. Common language runtime management includes, but is not limited to, three major activities: type control, structured exception handling, and garbage collection. *Type control* involves the verification and conversion of item types during execution. *Managed exception handling* is functionally similar to "unmanaged" structured exception handling, but it is performed by the runtime rather than by the operating system. *Garbage collection* involves the automatic identification and disposal of objects no longer in use.

A .NET application, intended for the common language runtime environment, consists of one or more *managed executables*, each of which carries metadata and (optionally) managed code. Managed code is optional because it is always possible to build a managed executable containing no methods. (Obviously, such an executable can be used only as an auxiliary part of an application.) Managed .NET applications are called *assemblies*. (This statement is somewhat

simplified; for more details about assemblies, application domains, and applications, see Chapter 6.) The managed executables are referred to as *modules*. You can create single-module assemblies and multimodule assemblies. As illustrated in Figure 1-1, each assembly contains one prime module, which carries the assembly identity information in its metadata.

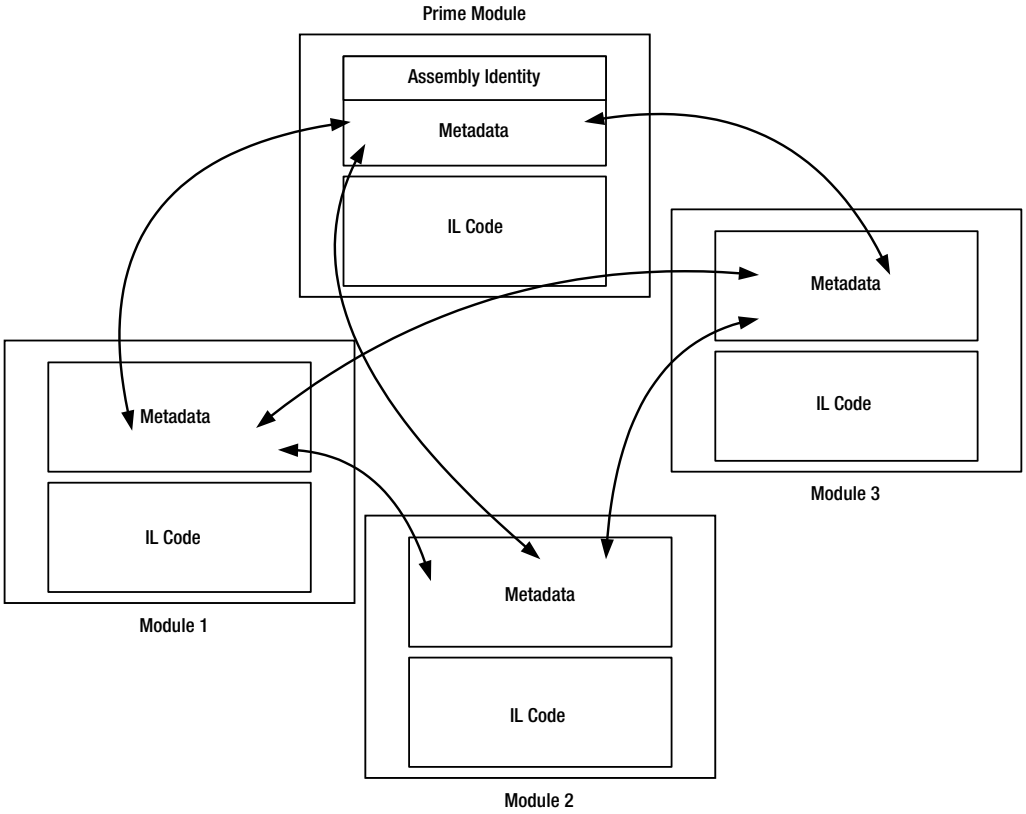

Figure 1-1. *A multimodule .NET assembly*

Figure 1-1 also shows that the two principal components of a managed executable are the metadata and the IL code. The two major common language runtime subsystems dealing with each component are, respectively, the loader and the just-in-time (JIT) compiler.

In brief, the *loader* reads the metadata and creates in memory an internal representation and layout of the classes and their members. It performs this task on demand, meaning a class is loaded and laid out only when it is referenced. Classes that are never referenced are never loaded. When loading a class, the loader runs a series of consistency checks of the related metadata.

The *JIT compiler*, relying on the results of the loader's activity, compiles the methods encoded in IL into the native code of the underlying platform. Because the runtime is not an interpreter, it does not execute the IL code. Instead, the IL code is compiled in memory into the native code, and the native code is executed. The JIT compilation is also done on demand, meaning a method is compiled only when it is called. The compiled methods stay cached in memory. If memory is limited, however, as in the case of a small computing device such as a

handheld PDA or a smart phone, the methods can be discarded if not used. If a method is called again after being discarded, it is recompiled.

Figure 1-2 illustrates the sequence of creating and executing a managed .NET application. Arrows with hollow circles at the base indicate data transfer; the arrow with the black circle represents requests and control messages.

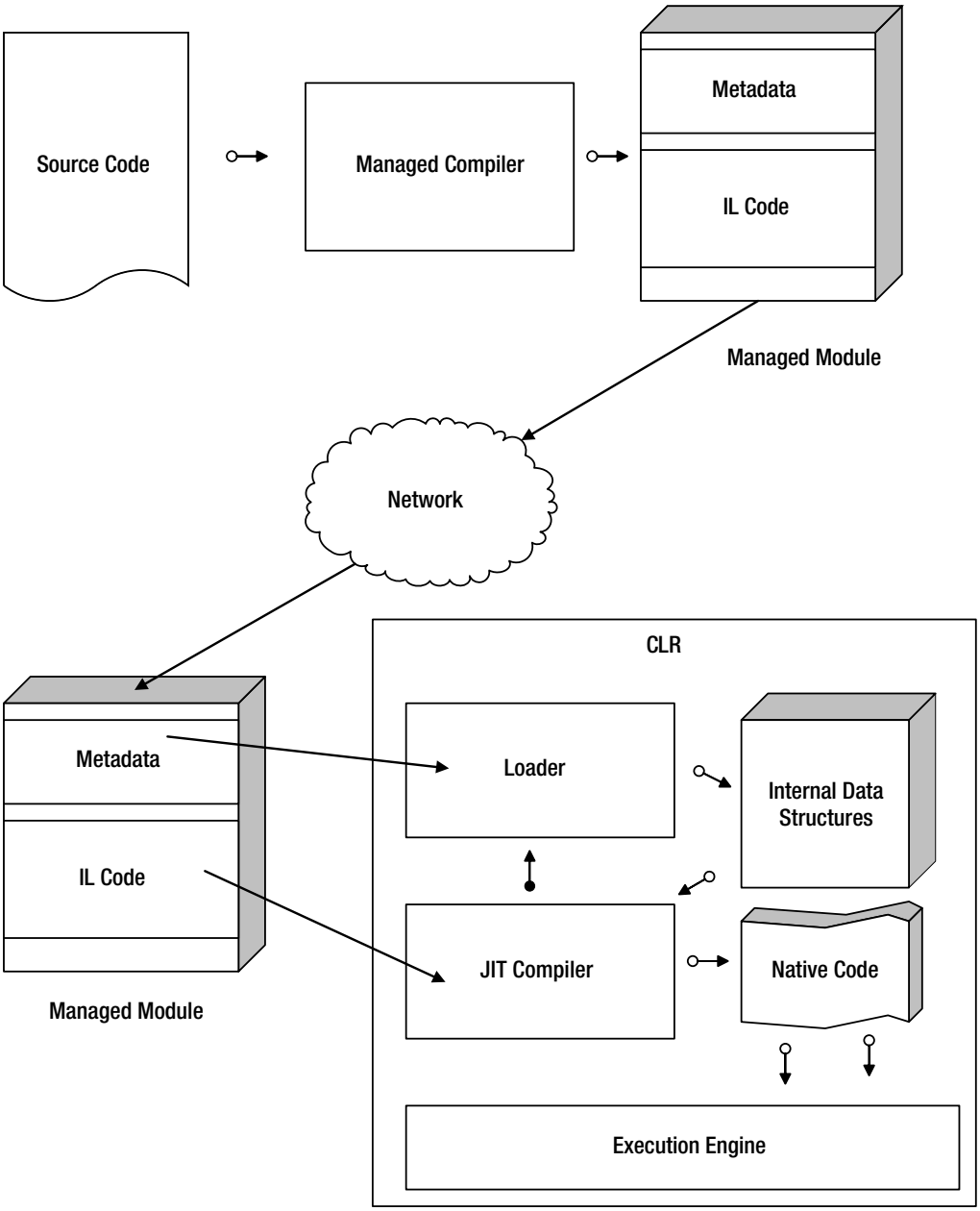

Figure 1-2. *The creation and execution of a managed .NET application*

You can precompile a managed executable from IL to the native code using the NGEN utility. You can do this when the executable is expected to run repeatedly from a local disk in order to save time on JIT compilation. This is standard procedure, for example, for managed components of the .NET Framework, which are precompiled during installation. (Tom Archer refers to this as *install-time code generation*.) In this case, the precompiled code is saved to the local disk or other storage, and every time the executable is invoked, the precompiled native-code version is used instead of the original IL version. The original file, however, must also be present because the precompiled version must be authenticated against the original file before it is allowed to execute.

With the roles of the metadata and the IL code established, I'll now cover the ways you can use ILAsm to describe them.

Simple Sample: The Code

No, the sample will not be "Hello, world!" This sample is a simple managed console application that prompts the user to enter an integer and then identifies the integer as odd or even. When the user enters something other than a decimal number, the application responds with "How rude!" and terminates. (See the source file Simple.il on the Apress Web site at `http://www.apress.com`.)

The sample, shown in Listing 1-1, uses managed console APIs from the .NET Framework class library for console input and output, and it uses the unmanaged function `sscanf` from the C run-time library for input string conversion to an integer.

■ **Note** To increase code readability throughout this book, all ILAsm keywords within the code listings appear in bold.

Listing 1-1. *OddOrEven Sample Application*

```
//----------- Program header
.assembly extern mscorlib { auto }
.assembly OddOrEven  { }
.module OddOrEven.exe
//----------- Class declaration
.namespace Odd.or {
    .class public auto ansi Even extends [mscorlib]System.Object {
//----------- Field declaration
        .field public static int32 val
//----------- Method declaration
        .method public static void check( ) cil managed {
            .entrypoint
            .locals init (int32 Retval)
        AskForNumber:
            ldstr "Enter a number"
            call void [mscorlib]System.Console::WriteLine(string)
```

```
            call string [mscorlib]System.Console::ReadLine ()
            ldsflda valuetype CharArray8 Format
            ldsflda int32 Odd.or.Even::val
            call vararg int32 sscanf(string,int8*,...,int32*)
            stloc Retval
            ldloc Retval
            brfalse Error
            ldsfld int32 Odd.or.Even::val
            ldc.i4 1
            and
            brfalse ItsEven
            ldstr "odd!"
            br PrintAndReturn
        ItsEven:
            ldstr "even!"
            br PrintAndReturn
        Error:
            ldstr "How rude!"
        PrintAndReturn:
            call void [mscorlib]System.Console::WriteLine(string)
            ldloc Retval
            brtrue AskForNumber
            ret
        } // End of method
    } // End of class
} // End of namespace
//----------- Global items
.field public static valuetype CharArray8 Format at FormatData
//----------- Data declaration
.data FormatData = bytearray(25 64 00 00 00 00 00 00) // % d . . . . . .
//----------- Value type as placeholder
.class public explicit CharArray8
              extends [mscorlib]System.ValueType { .size 8  }
//----------- Calling unmanaged code
.method public static pinvokeimpl("msvcrt.dll" cdecl)
    vararg int32 sscanf(string,int8*) cil managed { }
```

In the following sections, I'll walk you through this source code line by line.

Program Header

This is the program header of the OddOrEven application:

```
.assembly extern mscorlib { auto }
.assembly OddOrEven  { }
.module OddOrEven.exe
```

`.assembly extern mscorlib { auto }` defines a metadata item named `Assembly Reference` (or `AssemblyRef`), identifying the external managed application (assembly) used in this program. In this case, the external application is Mscorlib.dll, the main assembly of the .NET Framework classes. (The topic of the .NET Framework class library itself is beyond the scope of this book; for further information, consult the detailed specification of the .NET Framework class library published as Partition IV of the Ecma International/ISO standard.)

The Mscorlib.dll assembly contains declarations of all the base classes from which all other classes are derived. Although theoretically you could write an application that never uses anything from Mscorlib.dll, I doubt that such an application would be of any use. (One obvious exception is Mscorlib.dll itself.) Thus, it's a good habit to begin a program in ILAsm with a declaration of `AssemblyRef` to Mscorlib.dll, followed by declarations of other `AssemblyRef`s (if any).

The scope of an `AssemblyRef` declaration (between the curly braces) can contain additional information identifying the referenced assembly, such as the version or culture (previously known as `locale`). Because this information is not relevant to understanding this sample, I have omitted it here. (Chapter 5 describes this additional information in detail.) Instead, I used the keyword `auto`, which prompts ILASM to automatically discover the latest version of the referenced assembly.

Note that the assembly autodetection feature is specific to ILASM 2.0 and newer. Versions 1.0 and 1.1 have no autodetection, but they allow referencing Mscorlib.dll (and only it) without additional identifying information. So when using older versions of ILASM, just leave the `AssemblyRef` scope empty.

Note also that although the code references the assembly Mscorlib.dll, `AssemblyRef` is declared by filename only, without the extension. Including the extension causes the loader to look for Mscorlib.dll.dll or Mscorlib.dll.exe, resulting in a run-time error.

`.assembly OddOrEven { }` defines a metadata item named `Assembly`, which, to no one's surprise, identifies the current application (assembly). Again, you could include additional information identifying the assembly in the assembly declaration—see Chapter 6 for details—but it is not necessary here. Like `AssemblyRef`, the assembly is identified by its filename, without the extension.

Why do you need to identify the application as an assembly? If you don't, it will not be an application at all; rather, it will be a nonprime module—part of some other application (assembly)—and as such will not be able to execute on its own. Giving the module an .exe extension changes nothing; only assemblies can be executed.

`.module OddOrEven.exe` defines a metadata item named `Module`, identifying the current module. Each module, prime or otherwise, carries this identification in its metadata. Note that the module is identified by its full filename, including the extension. The path, however, must not be included.

Class Declaration

This is the class declaration of the OddOrEven application:

```
.namespace Odd.or {
    .class public auto ansi Even extends [mscorlib]System.Object {
        ...
    }
    ...
}
```

`.namespace Odd.or { … }` declares a namespace. A namespace does not represent a separate metadata item. Rather, a namespace is a common prefix of the full names of all the classes declared within the scope of the namespace declaration.

`.class public auto ansi Even extends [mscorlib]System.Object { ... }` defines a metadata item named `Type Definition` (`TypeDef`). Each class, structure, or enumeration defined in the current module is described by a respective `TypeDef` record in the metadata. The name of the class is `Even`. Because it is declared within the scope of the namespace `Odd.or`, its full name (by which it can be referenced elsewhere and by which the loader identifies it) is `Odd.or.Even`. You could forgo the namespace declaration and just declare the class by its full name; it would not make any difference.

The keywords `public`, `auto`, and `ansi` define the flags of the `TypeDef` item. The keyword `public`, which defines the visibility of the class, means the class is visible outside the current assembly. (Another keyword for class visibility is `private`, the default, which means the class is for internal use only and cannot be referenced from outside.)

The keyword `auto` in this context defines the class layout style (automatic, the default), directing the loader to lay out this class however it sees fit. Alternatives are `sequential` (which preserves the specified sequence of the fields) and `explicit` (which explicitly specifies the offset for each field, giving the loader exact instructions for laying out the class).

The keyword `ansi` defines the mode of string conversion within the class when interoperating with the unmanaged code. This keyword, the default, specifies that the strings will be converted to and from "normal" C-style strings of bytes. Alternative keywords are `unicode` (strings are converted to and from UTF-16 Unicode) and `autochar` (the underlying platform determines the mode of string conversion).

The clause `extends [mscorlib]System.Object` defines the parent, or base class, of the class `Odd.or.Even`. The code `[mscorlib]System.Object` represents a metadata item named `Type Reference` (`TypeRef`). This particular `TypeRef` has `System` as its namespace, `Object` as its name, and `AssemblyRef mscorlib` as the resolution scope. Each class defined outside the current module is addressed by `TypeRef`. You can also address the classes defined in the current module by `TypeRef`s instead of `TypeDef`s, which is considered harmless enough but not nice.

By default, all classes are derived from the class `System.Object` defined in the assembly Mscorlib.dll. Only `System.Object` itself and the interfaces have no base class, as explained in Chapter 7.

The structures—referred to as *value types* in .NET lingo—are derived from the `[mscorlib]System.ValueType` class. The enumerations are derived from the `[mscorlib]System.Enum` class. Because these two distinct kinds of `TypeDef`s are recognized solely by the classes they extend, you must use the `extends` clause every time you declare a value type or an enumeration.

You have probably noticed that the declaration of `TypeDef` in the sample contains three default items: the flags `auto` and `ansi` and the `extends` clause. Yes, in fact, I could have declared the same `TypeDef` as `.class public Even { ... }`, but then I would not be able to discuss the `TypeDef` flags and the `extends` clause.

Finally, I must emphasize one important fact about the class declaration in ILAsm. (Please pay attention, and don't say I haven't told you!) Some languages require that all of a class's attributes and members be defined within the lexical scope of the class, defining the class as a whole in one place. In ILAsm the class needn't be defined all in one place.

In ILAsm, you can declare a `TypeDef` with some of its attributes and members, close the `TypeDef`'s scope, and then reopen the same `TypeDef` later in the source code to declare more of its attributes and members. This technique is referred to as *class amendment*.

When you amend a TypeDef, the flags, the extends clause, and the implements clause (not discussed here in the interests of keeping the sample simple) are ignored. You should define these characteristics of a TypeDef the first time you declare it.

There is no limitation on the number of TypeDef amendments or on how many source files a TypeDef declaration might span. You are required, however, to completely define a TypeDef within one module. Thus, it is impossible to amend the TypeDefs defined in other assemblies or other modules of the same assembly.

Chapter 7 provides detailed information about ILAsm class declarations.

USING PSEUDOFLAGS TO DECLARE A VALUE TYPE AND AN ENUMERATION

You might want to know about a little cheat that will allow you to circumvent the necessity of repeating the extends clause. ILAsm has two keywords, value and enum, that can be placed among the class flags to identify, respectively, value types and enumerations if you omit the extends clause. (If you include the extends clause, these keywords are ignored.) This is, of course, not a proper way to represent the metadata, because it can give the incorrect impression that value types and enumerations are identified by certain TypeDef flags. I am ashamed that ILAsm contains such lowly tricks, but I am too lazy to type extends [mscorlib]System.ValueType again and again. ILDASM never resorts to these cheats and always truthfully prints the extends clause, but ILDASM has the advantage of being a software utility.

Field Declaration

This is the field declaration of the OddOrEven application:

```
.field public static int32 val
```

.field public static int32 val defines a metadata item named Field Definition (FieldDef). Because the declaration occurs within the scope of class Odd.or.Even, the declared field belongs to this class.

The keywords public and static define the flags of the FieldDef. The keyword public identifies the accessibility of this field and means the field can be accessed by any member for whom this class is visible. Alternative accessibility flags are as follows:

- The assembly flag specifies that the field can be accessed from anywhere within this assembly but not from outside.
- The family flag specifies that the field can be accessed from any of the classes descending from Odd.or.Even.
- The famandassem flag specifies that the field can be accessed from any of those descendants of Odd.or.Even that are defined in this assembly.
- The famorassem flag specifies that the field can be accessed from anywhere within this assembly as well as from any descendant of Odd.or.Even, even if the descendant is declared outside this assembly.
- The private flag specifies that the field can be accessed from Odd.or.Even only.

- The `privatescope` flag specifies that the field can be accessed from anywhere within current module. This flag is the default. The `privatescope` flag is a special case, and I strongly recommend you do not use it. Private scope items are exempt from the requirement of having a unique parent/name/signature triad, which means you can define two or more private scope items within the same class that have the same name and the same type. Some compilers emit private scope items for their internal purposes. It is the compiler's problem to distinguish one private scope item from another; if you decide to use private scope items, you should at least give them unique names. Because the default accessibility is `privatescope`, which can be a problem, it's important to remember to specify the accessibility flags.

The keyword `static` means the field is static—that is, it is shared by all instances of class `Odd.or.Even`. If you did not designate the field as static, it would be an instance field, individual to a specific instance of the class.

The keyword `int32` defines the type of the field, a 32-bit signed integer. (Chapter 8 describes types and signatures.) And, of course, `val` is the name of the field.

You can find a detailed explanation of field declarations in Chapter 9.

Method Declaration

This is the method declaration of the OddOrEven application:

```
    .method public static void check( ) cil managed {
        .entrypoint
        .locals init (int32 Retval)
...
}
```

`.method public static void check( ) cil managed { ... }` defines a metadata item named `Method Definition` (`MethodDef`). Because it is declared within the scope of `Odd.or.Even`, this method is a member of this class.

The keywords `public` and `static` define the flags of `MethodDef` and mean the same as the similarly named flags of `FieldDef` discussed in the preceding section. Not all the flags of `FieldDef`s and `MethodDef`s are identical—see Chapter 9 as well as Chapter 10 for details—but the accessibility flags are, and the keyword `static` means the same for fields and methods.

The keyword `void` defines the return type of the method. If the method had a calling convention that differed from the default, you would place the respective keyword after the flags but before the return type. Calling convention, return type, and types of method parameters define the signature of the `MethodDef`. Note that a lack of parameters is expressed as `( )`, never as `(void)`. The notation `(void)` would mean that the method has one parameter of type `void`, which is an illegal signature.

The keywords `cil` and `managed` define so-called implementation flags of the `MethodDef` and indicate that the method body is represented in IL. A method represented in native code rather than in IL would carry the implementation flags `native unmanaged`.

Now, let's proceed to the method body. In ILAsm, the method body (or method scope) generally contains three categories of items: instructions (compiled into IL code), labels marking the instructions, and directives (compiled into metadata, header settings, managed exception handling clauses, and so on—in short, anything but IL code). Outside the method body, only directives exist. Every declaration discussed so far has been a directive.

`.entrypoint` identifies the current method as the entry point of the application (the assembly). Each managed EXE file must have a single entry point. The ILAsm compiler will refuse to compile a module without a specified entry point, unless you use the */DLL* command-line option.

`.locals init (int32 Retval)` defines the single local variable of the current method. The type of the variable is `int32`, and its name is `Retval`. The keyword `init` means the local variables will be initialized at run time before the method executes. If the local variables are not designated with this keyword in even one of the assembly's methods, the assembly will fail verification (in a security check performed by the common language runtime) and will be able to run only in full-trust mode, when verification is disabled. For that reason, you should never forget to use the keyword `init` with the local variable declaration. If you need more than one local variable, you can list them, separated by commas, within the parentheses—for example, `.locals init (int32 Retval, string TempStr)`.

```
AskForNumber:
    ldstr "Enter a number"
    call void [mscorlib]System.Console::WriteLine(string)
```

`AskForNumber:` is a label. It needn't occupy a separate line; the IL disassembler marks every instruction with a label on the same line as the instruction. Labels are not compiled into metadata or IL; rather, they are used solely for the identification of certain offsets within IL code at compile time.

A label marks the first instruction that follows it. Labels don't mark directives. In other words, if you moved the `AskForNumber` label two lines up so that the directives `.entrypoint` and `.locals` separated the label and the first instruction, the label would still mark the first instruction.

An important note before I go on to the instructions: IL is strictly a stack-based language. Every instruction takes something (or nothing) from the top of the stack and puts something (or nothing) onto the stack. Some instructions have parameters in addition to arguments and some don't, but the general rule does not change: instructions take all required arguments (if any) from the stack and put the results (if any) onto the stack. No IL instruction can address a local variable or a method parameter directly, except the instructions of `load` and `store` groups, which, respectively, put the value or the address of a variable or a parameter onto the stack or take the value from the stack and put it into a variable or a parameter.

Elements of the IL stack are not bytes or words, but slots. When I talk about IL stack depth, I am talking in terms of items put onto the stack, with no regard for the size of each item. Each slot of the IL stack carries information about the type of its current "occupant." And if you put an `int32` item on the stack and then try to execute an instruction, which expects, for instance, a string, the JIT compiler becomes very unhappy and very outspoken, throwing an `Unexpected Type` exception and aborting the compilation.

`ldstr "Enter a number"` is an instruction that creates a string object from the specified string constant and loads a reference to this object onto the stack. The string constant in this case is stored in the metadata. You can refer to such strings as *common language runtime string constants* or *metadata string constants*. You can store and handle the string constants in another way, as explained in a few moments, but `ldstr` deals exclusively with common language runtime string constants, which are always stored in Unicode (UTF-16) format.

`call void [mscorlib]System.Console::WriteLine(string)` is an instruction that calls a console output method from the .NET Framework class library. The string is taken from the stack as the method argument, and nothing is put back, because the method returns `void`.

The parameter of this instruction is a metadata item named `Member Reference (MemberRef)`. It refers to the static method named `WriteLine`, which has the signature `void(string)`; the method is a member of class `System.Console`, declared in the external assembly `mscorlib`. The `MemberRefs` are members of `TypeRefs`—discussed earlier in this chapter in the section "Class Declaration"—just as `FieldDefs` and `MethodDefs` are `TypeDef` members. However, there are no separate `FieldRefs` and `MethodRefs`; the `MemberRefs` cover references to both fields and methods.

You can distinguish field references from method references by their signatures. `MemberRefs` for fields and for methods have different calling conventions and different signature structures. Chapter 8 discusses signatures, including those of `MemberRefs`, in detail.

How does the IL assembler know what type of signature should be generated for a `MemberRef`? Mostly from the context. For example, if a `MemberRef` is the parameter of a `call` instruction, it must be a `MemberRef` for a method. In certain cases in which the context is not clear, the compiler requires explicit specifications, such as `method void Odd.or.Even::check( )` or `field int32 Odd.or.Even::val`.

```
call string [mscorlib]System.Console::ReadLine()
ldsflda valuetype CharArray8 Format
ldsflda int32 Odd.or.Even::val
call vararg int32 sscanf(string,int8*,...,int32*)
```

`call string [mscorlib]System.Console::ReadLine()` is an instruction that calls a console input method from the .NET Framework class library. Nothing is taken from the stack, and a string is put onto the stack as a result of this call.

`ldsflda valuetype CharArray8 Format` is an instruction that loads the address of the static field `Format` of type `valuetype CharArray8`. (Both the field and the value type are declared later in the source code and are discussed in later sections.) IL has separate instructions for loading instance and static fields (`ldfld` and `ldsfld`) or their addresses (`ldflda` and `ldsflda`). Also note that the "address" loaded onto the stack is not exactly an address (or a C/C++ pointer) but rather a reference to the item (a field in this sample).

As you probably guessed, `valuetype CharArray8 Format` is another `MemberRef`, this time to the field `Format` of type `valuetype CharArray8`. Because this `MemberRef` is not attributed to any `TypeRef`, it must be a global item. (The following section discusses declaring global items.) In addition, this `MemberRef` is not attributed to any external resolution scope, such as `[mscorlib]`. Hence, it must be a global item defined somewhere in the current module.

`ldsflda int32 Odd.or.Even::val` is an instruction that loads the address of the static field `val`, which is a member of the class `Odd.or.Even`, of type `int32`. But because the method being discussed is also a member of `Odd.or.Even`, why do you need to specify the full class name when referring to a member of the same class? Such are the rules of ILAsm: all references must be fully qualified. It might look a bit cumbersome, compared to most high-level languages, but it has its advantages. You don't need to keep track of the context, and all references to the same item look the same throughout the source code. And the IL assembler doesn't need to load and inspect the referenced assemblies to resolve the ambiguous references, which means the IL assembler can compile a module in the absence of the referenced assemblies and

modules (if you are not using the autodetection of the referenced assemblies, which, of course, doesn't work without the assemblies to detect).

Because both class `Odd.or.Even` and its field `val` are declared in the same module, the ILAsm compiler will not generate a `MemberRef` item but instead will use a `FieldDef` item. This way there will be no need to resolve the reference at run time.

`call vararg int32 sscanf(string,int8*,...,int32*)` is an instruction that calls the global static method `sscanf`. This method takes three items currently on the stack (the string returned from `System.Console::ReadLine`, the reference to the global field `Format`, and the reference to the field `Odd.or.Even::val`) and puts the result of type `int32` onto the stack.

This method call has two major peculiarities. First, it is a call to an unmanaged method from the C run-time library. I'll defer the explanation of this issue until I discuss the declaration of this method. (I have a formal excuse for that because, after all, at the call site managed and unmanaged methods look the same.)

The second peculiarity of this method is its calling convention, `vararg`, which means this method has a variable argument list. The `vararg` methods have some (or no) mandatory parameters, followed by an unspecified number of optional parameters of unspecified types—unspecified, that is, at the moment of the method declaration. When the method is invoked, all the mandatory parameters (if any) plus all the optional parameters used in this invocation (if any) should be explicitly specified.

Let's take a closer look at the list of arguments in this call. The ellipsis refers to a pseudoargument of a special kind, known as a *sentinel*. A sentinel's role can be formulated as "separating the mandatory arguments from the optional ones," but I think it would be less ambiguous to say that a sentinel immediately precedes the optional arguments and it is a prefix of the optional part of a `vararg` signature.

What is the difference? An ironclad common language runtime rule concerning the `vararg` method signatures dictates that a sentinel cannot be used when no optional arguments are specified. Thus, a sentinel can never appear in `MethodDef` signatures—only mandatory parameters are specified when a method is declared—and it should not appear in call site signatures when only mandatory arguments are supplied. Signatures containing a trailing sentinel are illegal. That's why I think it is important to look at a sentinel as the beginning of optional arguments and not as a separator between mandatory and optional arguments or (heaven forbid!) as the end of mandatory arguments.

For those less familiar with the C runtime, I should note that the function `sscanf` parses and converts the buffer string (the first argument) according to the format string (the second argument), puts the results in the rest of the pointer arguments, and returns the number of successfully converted items. In this sample, only one item will be converted, so `sscanf` will return 1 on success or 0 on failure.

```
stloc Retval
ldloc Retval
brfalse Error
```

`stloc Retval` is an instruction that takes the result of the call to `sscanf` from the stack and stores it in the local variable `Retval`. You need to save this value in a local variable because you will need it later.

`ldloc Retval` copies the value of `Retval` back onto the stack. You need to check this value, which was taken off the stack by the `stloc` instruction.

`brfalse Error` takes an item from the stack, and if it is 0, it branches (switches the computation flow) to the label `Error`.

```
ldsfld int32 Odd.or.Even::val
ldc.i4 1
and
brfalse ItsEven
ldstr "odd!"
br PrintAndReturn
```

`ldsfld int32 Odd.or.Even::val` is an instruction that loads the value of the static field `Odd.or.Even::val` onto the stack. If the code has proceeded this far, the string-to-integer conversion must have been successful, and the value that resulted from this conversion must be sitting in the field `val`. The last time you addressed this field, you used the instruction `ldsflda` to load the field address onto the stack. This time you need the value, so you use `ldsfld`.

`ldc.i4 1` is an instruction that loads the constant `1` of type `int32` onto the stack.

Instruction `and` takes two items from the stack—the value of the field `val` and the integer constant `1`—performs a bitwise AND operation and puts the result onto the stack. Performing the bitwise AND operation with `1` zeroes all the bits of the value of `val` except the least-significant bit.

`brfalse ItsEven` takes an item from the stack (the result of the bitwise AND operation), and if it is 0, it branches to the label `ItsEven`. The result of the previous instruction is 0 if the value of `val` is even, and it is 1 if the value is odd.

`ldstr "odd!"` is an instruction that loads the string `odd!` onto the stack.

`br PrintAndReturn` is an instruction that does not touch the stack and branches unconditionally to the label `PrintAndReturn`.

The rest of the code in the `Odd.or.Even::check` method should be clear. This section has covered all the instructions used in this method except `ret`, which is fairly obvious: it returns whatever is on the stack. If the method's return type does not match the type of the item on the stack, the JIT compiler will disapprove, throw an exception, and abort the compilation. It will do the same if the stack contains more than one item by the time `ret` is reached or if the method is supposed to return `void` (that is, not return anything) and the stack still contains an item—or, conversely, if the method is supposed to return something and the stack is empty.

Global Items

These are the global items of the OddOrEven application:

```
{
...
} // End of namespace
.field public static valuetype CharArray8 Format at FormatData
```

`.field public static valuetype CharArray8 Format at FormatData` declares a static field named `Format` of type `valuetype CharArray8`. As you might remember, you used a reference to this field in the method `Odd.or.Even::check`.

This field differs from, for example, the field `Odd.or.Even::val` because it is declared outside any class scope and hence does not belong to any class. It is thus a global item. Global items belong to the module containing their declarations. As you've learned, a module is a

managed executable file (EXE or DLL); one or more modules constitute an assembly, which is the primary building block of a managed .NET application; and each assembly has one prime module, which carries the assembly identification information in its metadata.

Actually, a little trick is connected with the concept of global items not belonging to any class. In fact, the metadata of every module contains one special `TypeDef` named `<Module>`, which represents…any guesses? Yes, you are absolutely right.

This `TypeDef` is always present in the metadata, and it always holds the honorable first position in the `TypeDef` table. However, `<Module>` is not a proper `TypeDef`, because its attributes are limited compared to "normal" `TypeDef`s (classes, value types, and so on). This sounds almost like real life—the more honorable the position you hold, the more limited your options are.

`<Module>` cannot be public, that is, visible outside its assembly. `<Module>` can have only static members, which means all global fields and methods must be static. In addition, `<Module>` cannot have events or properties because events and properties cannot be static. (Consult Chapter 15 for details.) The reason for this limitation is obvious: given that an assembly always contains exactly one instance of every module, the concept of instantiation becomes meaningless.

The accessibility of global fields and methods differs from the accessibility of member fields and methods belonging to a "normal" class. Even public global items cannot be accessed from outside the assembly. `<Module>` does not extend anything—that is, it has no base class—and no class can inherit from `<Module>`. However, all the classes declared within a module have full access to the global items of this module, including the private ones.

This last feature is similar to class nesting and is quite different from class inheritance. (Derived classes don't have access to the private items of their base classes.) A *nested class* is a class declared within the scope of another class. That other class is usually referred to as an *enclosing class* or an *encloser*. A nested class is not a member class or an inner class in the sense that it has no implicit access to the encloser's instance reference (*this*). A nested class is connected to its encloser by three facts only: it is declared within the encloser's lexical scope; its visibility is "filtered" by the encloser's visibility (that is, if the encloser is private, the nested class will not be visible outside the assembly, regardless of its own visibility); and it has access to all of the encloser's members.

Because all the classes declared within a module are by definition declared within the lexical scope of the module, it is only logical that the relationship between the module and the classes declared in it is that of an encloser and nested classes.

As a result, global item accessibilities `public`, `assembly`, and `famorassem` all amount to `assembly`; `private`, `family`, and `famandassem` amount to `private`; and `privatescope` is, well, `privatescope`. The metadata validity rules explicitly state that only three accessibilities are permitted for the global fields and methods: `public` (which is actually `assembly`), `private`, and `privatescope`. The loader, however, is more serene about the accessibility flags of the global items: it allows any accessibility flags to be set, interpreting them as just described (as `assembly`, `private`, or `privatescope`).

Mapped Fields

This is the mapped field of the OddOrEven application:

```
.field public static valuetype CharArray8 Format at FormatData
```

The declaration of the field `Format` contains one more new item, the clause `at FormatData`. This clause indicates the `Format` field is located in the data section of the module and its location is identified by the data label `FormatData`. (The following section discusses data declaration and labeling.)

Compilers widely use this technique of mapping fields to data for field initialization. This technique does have some limitations, however. First, mapped fields must be static. This is logical. After all, the mapping itself is static, because it takes place at compile time. And even if you manage to map an instance field, all the different instances of this field will be physically mapped to the same memory, which means you'll wind up with a static field anyway. Because the loader, encountering a mapped instance field, decides in favor of "instanceness" and completely ignores the field mapping, the mapped instance fields are laid out just like all other instance fields.

Second, the mapped fields belong in the data section and hence are unreachable for the garbage collection subsystem of the common language runtime, which automatically disposes of unused objects. For this reason, mapped fields cannot be of a type that is subject to garbage collection (such as `class` or `array`). Value types are permitted as types of the mapped fields, as long as these value types have no members of types that are subject to garbage collection. If this rule is violated, the loader throws a `Type Load` exception and aborts loading the module.

Third, mapping a field to a predefined memory location leaves this field wide open to access and manipulation. This is perfectly fine from the point of view of security as long as the field does not have an internal structure whose parts are not intended for public access. That's why the type of a mapped field cannot be any value type that has nonpublic member fields. The loader enforces this rule strictly and checks for nonpublic fields all the way down. For example, if the type of a mapped field is value type A, the loader will check whether its fields are all public. If among these fields is one field of value type B, the loader will check whether value type B's fields are also all public. If among these fields are two fields of value types C and D—well, you get the picture. If the loader finds a nonpublic field at any level in the type of a mapped field, it throws a `Type Load` exception and aborts the loading.

Data Declaration

This is the data declaration of the OddOrEven application:

```
.field public static valuetype CharArray8 Format at FormatData
.data FormatData = bytearray(25 64 00 00 00 00 00 00)
```

`.data FormatData = bytearray(25 64 00 00 00 00 00 00)` defines a data segment labeled `FormatData`. This segment is 8 bytes long and has ASCII codes of the characters *%* (0x25) and *d* (0x64) in the first 2 bytes and zeros in the remaining 6 bytes.

The segment is described as `bytearray`, which is the most ubiquitous way to describe data in ILAsm. The numbers within the parentheses represent the hexadecimal values of the bytes, without the 0x prefix. The byte values should be separated by spaces, and I recommend you always use the two-digit form, even if one digit would suffice (as in the case of 0, for example).

It is fairly obvious you can represent literally any data as a `bytearray`. For example, instead of using the quoted string in the instruction `ldstr "odd!"`, you could use a `bytearray` presentation of the string:

```
ldstr bytearray(6F 00 64 00 64 00 21 00 00 00)
```

The numbers in parentheses represent the Unicode characters *o*, *d*, *d*, and *!* and the zero terminator. When you use ILDASM, you can see `bytearrays` everywhere. A `bytearray` is a universal, type-neutral form of data representation, and ILDASM uses it whenever it cannot identify the type associated with the data as one of the elementary types, such as `int32`.

On the other hand, you could define the data `FormatData` as follows:

```
.data FormatData = int64(0x0000000000006425)
```

This would result in the same data segment size and contents. When you specify a type declaring a data segment (for instance, `int64`), no record concerning this type is entered into metadata or anywhere else. The ILAsm compiler uses the specified type for two purposes only: to identify the size of the data segment being allocated and to identify the byte layout within this segment.

Value Type As Placeholder

This is the value type used as a placeholder:

```
.field public static valuetype CharArray8 Format at FormatData
.data FormatData = bytearray(25 64 00 00 00 00 00 00)
.class public explicit CharArray8
              extends [mscorlib]System.ValueType { .size 8  }
```

`.class public explicit CharArray8 extends [mscorlib]System.ValueType { .size 8 }` declares a value type that has no members but has an explicitly specified size, 8 bytes. Declaring such a value type is a common way to declare "just a piece of memory." In this case, you don't need to declare any members of this value type because you aren't interested in the internal structure of this piece of memory; you simply want to use it as a type of your global field `Format` to specify the field's size. In a sense, this value type is nothing but a placeholder.

Could you use an array of 8 bytes instead and save yourself the declaration of another value type? You could if you did not intend to map the field to the data. Because arrays are subject to garbage collection, they are not allowed as types of mapped fields.

Using value types as placeholders is popular with managed C/C++ compilers because of the need to store and address numerous ANSI string constants. The Visual C# and Visual Basic .NET compilers, which deal mostly with Unicode strings, are less enthusiastic about this technique because they can directly use the common language runtime string constants, which are stored in metadata in Unicode format.

Calling Unmanaged Code

This is how the OddOrEven application declares the unmanaged method, which is called from the managed method `check`:

```
.method public static pinvokeimpl("msvcrt.dll" cdecl)
    vararg int32 sscanf(string,int8*) cil managed { }
```

The line `.method public static pinvokeimpl("msvcrt.dll" cdecl) vararg int32 sscanf(string, int8*) cil managed { }` declares an unmanaged method, to be called from managed code. The attribute `pinvokeimpl("msvcrt.dll" cdecl)` indicates that this is an unmanaged method, called using the mechanism known as *platform invocation* or `P/Invoke`.

This attribute also indicates that this method resides in the unmanaged DLL Msvcrt.dll and has the calling convention `cdecl`. This calling convention means the unmanaged method handles the arguments the same way an ANSI C function does.

The method takes two mandatory parameters of types `string` and `int8*` (the equivalent of C/C++ `char*`) and returns `int32`. Being a `vararg` method, `sscanf` can take any number of optional parameters of any type, but as you know already, neither the optional parameters nor a sentinel is specified when a `vararg` method is declared.

Platform invocation is the mechanism the common language runtime provides to facilitate the calls from the managed code to unmanaged functions. Behind the scenes, the runtime constructs the so-called stub, or *thunk*, which allows the addressing of the unmanaged function and conversion of managed argument types to the appropriate unmanaged types and back. This conversion is known as *parameter marshaling*.

What is being declared here is not an actual unmanaged method to be called but a stub generated by the runtime, as it is seen from the managed code, which explains the implementation flags `cil managed`. Specifying the method signature as `int32(string, int8*)`, you specify the "managed side" of parameter marshaling. The unmanaged side of the parameter marshaling is defined by the actual signature of the unmanaged method being invoked.

The actual signature of the unmanaged function `sscanf` in C is `int sscanf(const char*, const char*, …)`. So, the first parameter is marshaled from managed type `string` to unmanaged type `char*`. Recall that when I declared the class `Odd.or.Even`, I specified the `ansi` flag, which means the managed strings by default are marshaled as ANSI C strings, that is, `char*`. And because the call to `sscanf` is made from a member method of class `Odd.or.Even`, you don't need to provide special information about marshaling the managed strings.

The second parameter of the `sscanf` declaration is `int8*`, which is a direct equivalent of `char*`; as a result, little marshaling is required. (ILAsm has type `char` as well, but it indicates a Unicode character rather than ANSI, equivalent to "unsigned short" in C, so you cannot use this type here.)

The optional parameters of the original (unmanaged) `sscanf` are supposed to be the pointers to items (variables) you want to fill while parsing the buffer string. The number and base types of these pointers are defined according to the format specification string (the second argument of `sscanf`). In this case, given the format specification string `"%d"`, `sscanf` will expect a single optional argument of type `int*`. When I call the managed thunk of `sscanf`, I provide the optional argument of type `int32*`, which might require marshaling to a native integer pointer only if you are dealing with a platform other than a 32-bit Intel platform (for example, an AMD or Intel 64-bit platform).

The `P/Invoke` mechanism is very useful because it gives you full access to rich and numerous native libraries and platform APIs. But don't overestimate the ubiquity of `P/Invoke`. Different platforms tend to have different APIs, so overtaxing `P/Invoke` can easily limit the portability of your applications. It's better to stick with the .NET Framework class library and take some consolation in the thought that by now you can make a fair guess about what lies at the bottom of this library.

Now that I've finished showing you the source code, find the sample file Simple.il on the Apress Web site, copy it into your working directory, compile it using the console command `ilasm simple` (assuming you have installed the .NET Framework and the Platform software development kit [SDK]), and try running the resulting Simple.exe.

Forward Declaration of Classes

This section is relevant only to earlier versions (1.0 and 1.1) of ILASM, but I still think it is useful information. Considering the size of the install base of the .NET Framework of these versions, chances are you will encounter older ILASM more than once.

If you have an older version of the .NET Framework installed, you can carry out a little experiment with the sample code. Open the source file Simple.il in any text editor, and modify it by moving the declaration of the value type `CharArray8` in front of the declaration of the field `Format`:

```
{
    ...
} // End of namespace
.class public explicit CharArray8
              extends [mscorlib]System.ValueType { .size 8  }
.field public static valuetype CharArray8 Format at FormatData
```

Everything seems to be in order. But when you try to recompile the file, ILAsm compilation fails with the error message `Unresolved MemberRef 'Format'`.

Now modify the source file again, this time moving the declaration of value type `CharArray8` before the declaration of the namespace `Odd.or`:

```
.class public explicit CharArray8
              extends [mscorlib]System.ValueType { .size 8  }
.namespace Odd.or {
    .class public auto ansi Even extends [mscorlib]System.Object {
        .field public static int32 val
        .method public static void check( ) cil managed {
            ...
            ldsflda valuetype CharArray8 Format
            ...
        } // End of method
    } // End of class
} // End of namespace
.field public static valuetype CharArray8 Format at FormatData
```

Now when you save the source code and try to recompile it, everything is back to normal. What's going on here?

After the first change, when the field `Format` was being referenced in the `ldsflda` instruction in the method `check`, the value type `CharArray8` had not been declared yet, so the respective `TypeRef` was emitted for it, and the signature of the field reference received the `TypeRef` as its type.

Then the value type `CharArray8` was declared, and a new `TypeDef` was created. After that, when the field `Format` was actually declared, its type was recognized as a locally declared value type, and the signature of the field definition received the `TypeDef` as its type. But, no field named `Format` with a `TypeRef` as its type was declared anywhere in this module. Hence, you get the reference-to-definition resolution failure.

(This is an inviting moment to criticize the ILAsm compiler's lack of ability to match the signatures on a pragmatic level, with type analysis and matching the `TypeRefs` to `TypeDefs` by full name and resolution scope. Have patience, however.)

After the second change in the source code, the value type `CharArray8` was declared first so that all references to it, no matter where they happen, refer to it as `TypeDef`. This is a rather obvious solution.

The solution becomes not so obvious when you consider two classes, members of which use each other's class as the type. Which class to declare first? Actually, both of them.

In the "Class Declaration" section I mentioned the class amendment technique, based on that ILAsm allows you to reopen a class scope to declare more class attributes and members. The general solution to the declaration/reference problem is to specify the empty-scope class definitions for all classes first. Following that, you can specify all the classes in full, with their attributes and members, as amendments. The "first wave" of class declarations should carry all class flags, `extends` clauses, and `implements` clauses and should include all nested classes (also with empty scopes). You should leave all the member declarations for later.

This technique of the forward declaration of classes guards against declaration/reference errors and, as a side effect, reduces the metadata size because it is unnecessary to emit redundant `TypeRefs` for locally defined classes.

(And the answer to the aforementioned criticism of the ILAsm compiler is that the compiler does signature matching in the fastest possible way, without needing more sophisticated and slower methods, as long as you use the class forward declaration.)

The need for the class forward declaration has been eliminated in version 2.0 of the ILAsm compiler.

Summary

This chapter touched briefly on the most important features of the common language runtime and ILAsm. You now know (in general terms) how the runtime functions, how a program in ILAsm is written, and how to define the basic components (classes, fields, and methods). You learned that the managed code can interoperate with the unmanaged (native) code and what the common language runtime is doing to facilitate this interoperation.

In the next chapter, you will continue working with the simple OddOrEven sample to learn some sophisticated features of the runtime and ILAsm.