CIL Programming: Under the Hood™ of .NET

JASON BOCK

CIL Programming: Under the Hood™ of .NET

ISBN: 1-59059-041-4
Printed and bound in the United States of America 12345678910

Technical Reviewer: Dan Fergus
Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell, Jason Gilmore, Karen Watterson, John Zukowski
Managing Editor: Grace Wong
Copy Editor: Ami Knox
Compositor: Diana Van Winkle, Van Winkle Design
Indexer: Carol Burbo
Artist and Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski
Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010
and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.
In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.
Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 9th Street, Suite 219, Berkeley, CA 94710.
Phone: 510-549-5930, Fax: 510-549-5939, Email: info@apress.com, Web site: http://www.apress.com

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

CHAPTER 6

.NET Languages and CIL

In this chapter, I'll walk through a number of code snippets written in various .NET languages and demonstrate what the differences are in their assemblies. I'll compare and contrast debug and release builds. You'll get a chance to look at how different language constructs are translated into CIL by the different compilers. I'll show you how a piece of code in one language may not create the output you expect. As you'll see throughout this chapter, what you code is not always what you get. By knowing CIL, you'll be able to figure out what's really going on.

Debug and Release Builds

To start, let's take a look at a small piece of code that's implemented in a couple of .NET languages and see what the CIL looks like. You'll build the code in both debug and release modes to find out how the CIL changes between the modes. Here's what the general flow of the code is doing in all of the implementations:

1. Gets the type of the current instance and stores it in a local variable called `someType`.

2. Gets the name of the type via the Name property, and stores it in a string called `typeName`.

3. If name is equal to "SimpleCode", does the following:

   - Declares an integer and call it `i`.

   - Creates a boolean called `yes` and set it to true.

   - Returns `yes`.

4. Returns a false value.

The C# Implementation

Here's what the pseudocode looks like in C#:

```
public bool TestForTypeName()
{
    Type someType = this.GetType();
    String typeName = someType.Name;

    if(true == typeName.Equals("SimpleCode"))
    {
        int i;
        bool yes = true;
        return yes;
    }
    return false;
}
```

Although the code follows the requirements to the letter, you as a developer may be squirming at three parts of the code:

- There's no reason to create `i`; it's a waste of space.
- The `yes` variable really isn't needed as you could simply return `true`.
- The `someType` variable really isn't needed either as it's never used after `Name` is called.

I don't know how many times I've seen dead code or code that could be optimized make it into the compilation process of a production system. This is usually due to a combination of a couple of issues—for example, the code has been updated by a number of developers, and with large functions it's not always obvious where the dead code lies.[1] In any event, let's see what C#'s compiler does with this method.

Listing 6-1 shows what the resulting CIL looks like if you compile `TestForTypeName()` in debug mode.

1. Technically, the C# compiler will tell you if a variable is not being used as is the case with `i`, but it won't be able to make the optimization with `someType`.

Listing 6-1. C# Compilation Results in Debug Mode

```
.method public hidebysig instance bool  TestForTypeName() cil managed
{
  // Code size       42 (0x2a)
  .maxstack  2
  .locals init ([0] class [mscorlib]System.Type someType,
           [1] string typeName,
           [2] int32 i,
           [3] bool yes,
           [4] bool CS$00000003$00000000)
  IL_0000:  ldarg.0
  IL_0001:  callinstance class [mscorlib]System.Type
    [mscorlib]System.Object::GetType()
  IL_0006:  stloc.0
  IL_0007:  ldloc.0
  IL_0008:  callvirt   instance string
    [mscorlib]System.Reflection.MemberInfo::get_Name()
  IL_000d:  stloc.1
  IL_000e:  ldloc.1
  IL_0414f:  ldstr      "SimpleCode"
  IL_0014:  callvirt   instance bool [mscorlib]System.String::Equals(string)
  IL_0019:  brfalse.s  IL_0022
  IL_001b:  ldc.i4.1
  IL_001c:  stloc.3
  IL_001d:  ldloc.3
  IL_001e:  stloc.s    CS$00000003$00000000
  IL_0020:  br.s       IL_0027
  IL_0022:  ldc.i4.0
  IL_0023:  stloc.s    CS$00000003$00000000
  IL_0025:  br.s       IL_0027
  IL_0027:  ldloc.s    CS$00000003$00000000
  IL_0029:  ret
} // end of method SimpleCode::TestForTypeName
```

Let me draw your attention to a couple of interesting things about the results. First, you'll see that all of the code has been translated into CIL and included in the assembly—that is, the C# compiler made no optimizations whatsoever. The other interesting aspect about the debug build is the fifth local variable, `CS$00000003$00000000`. It's not a variable you create in your code; the C# compiler creates this variable to make the debugging process "friendlier." To see what I mean by this statement, here are the last few lines of CIL with the C# code inlined:

```
//000022:         return yes;
  IL_001d:  ldloc.3
  IL_001e:  stloc.s    CS$00000003$00000000
  IL_0020:  br.s       IL_0027
//000023:       }
//000024:
//000025:       return false;
  IL_0022:  ldc.i4.0
  IL_0023:  stloc.s    CS$00000003$00000000
  IL_0025:  br.s       IL_0027
//000026:     }
  IL_0027:  ldloc.s    CS$00000003$00000000
  IL_0029:  ret
} // end of method SimpleCode::TestForTypeName
```

You'll notice that when each `return` statement is reached in C# code, there's no corresponding `ret` opcode. Instead, the return value is stored in `CS$00000003$00000000`, and then an unconditional branch occurs (`br.s`). This branch is made to the end of the method ("}"), where the value is finally returned.

Before I show why this has an advantage during debugging, let's tell the compiler to turn optimizations on for the debug build. You do this in VS .NET by right-clicking the project node in the Solutions Explorer window and selecting the Properties menu option. Select the Build node underneath Configuration Properties and set the Optimize Code property to true (see Figure 6-1 for details).

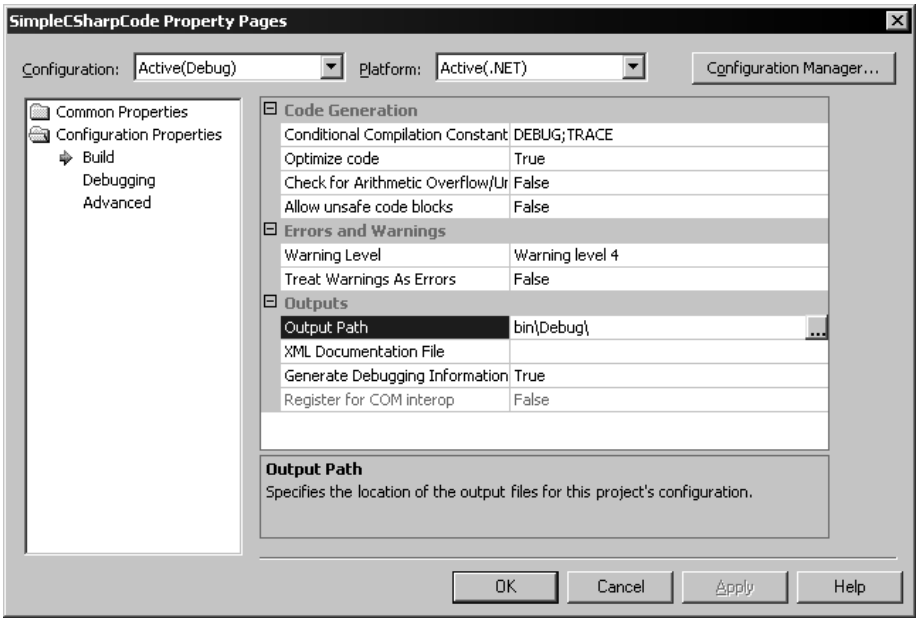

Figure 6-1. Project properties

When you recompile the code, the CIL will look like the code in Listing 6-2.

Listing 6-2. C# Compilation Results in Debug Mode with Optimizations

```
.method public hidebysig instance bool  TestForTypeName() cil managed
{
  // Code size       33 (0x21)
  .maxstack  2
  .locals init ([0] class [mscorlib]System.Type someType,
           [1] string typeName,
           [2] bool yes)
  IL_0000:  ldarg.0
  IL_0001:  call       instance class [mscorlib]System.Type
    [mscorlib]System.Object::GetType()
  IL_0006:  stloc.0
  IL_0007:  ldloc.0
  IL_0008:  callvirt   instance string
    [mscorlib]System.Reflection.MemberInfo::get_Name()
  IL_000d:  stloc.1
  IL_000e:  ldloc.1
  IL_000f:  ldstr      "SimpleCode"
  IL_0014:  callvirt   instance bool [mscorlib]System.String::Equals(string)
  IL_0019:  brfalse.s  IL_001f
  IL_001b:  ldc.i4.1
  IL_001c:  stloc.2
  IL_001d:  ldloc.2
  IL_001e:  ret
  IL_001f:  ldc.i4.0
  IL_0020:  ret
} // end of method SimpleCode::TestForTypeName
```

You can see that `i` is no longer declared as a local, and there's no temporary return value listed either. If you step through this optimized code in the debugger, you'll see that `i` doesn't show up in the Locals window as a local variable. You'll also notice that when you hit a `return` statement, you immediately jump out of the method, rather than go to the end.

If you're in debug mode, I'd strongly suggest not optimizing your build because of the changes that happen at the CIL level, especially when it comes to exiting a method. The reason is it gives you is a chance to see what the last value is before the method is finished. In the case of `TestForTypeName()`, it's not a big deal, because you can see in the code that you'll return a `true` or a `false`. This becomes a nice feature to have when you perform a calculation in the `return` statement.

To see why this feature is desirable, take a look at the following code:

```
public int IncrementIntValue()
{
    int i = 0;
    return i++;
}
```

If you turn on optimizations in debug mode, you'll end up never seeing what the value is for `i` unless you're in the calling method. If you want to see what `i` is before the method exits, just leave optimizations off.[2]

Now, when you compile the application in release mode, there's no debug file created, but the results are the same as before from a CIL perspective. The only difference is that the local variable names are mangled. Here's a snippet from `TestForTypeName()` in release mode with optimizations on:

```
.method public hidebysig instance bool
  TestForTypeName() cil managed
{
  // Code size       33 (0x21)
  .maxstack  2
  .locals init (class [mscorlib]System.Type V_0,
           string V_1,
           bool V_2)
```

The names you gave the variables are no longer there. This makes it a little harder to follow the code, as good variable names will give hints to people when they analyze decompiled code; they also make the symbol sizes smaller in the metadata, but they don't affect your code in any way.

The VB .NET Implementation

Now let's look at the method in VB .NET:

```
Public Function TestForTypeName() As Boolean
    Dim someType As Type = Me.GetType()
    Dim typeName As String = someType.Name

    If True = typeName.Equals("SimpleCode") Then
        Dim i As Integer
        Dim yes As Boolean = True
        Return yes
    End If

    Return False
End Function
```

2. In this case, it's easy to see that `i` will be 1 when the method is finished, but in more complex cases it may be nice to see the value before the method exits.

VB .NET is similar to C# in that the CIL results are the same in both debug and release mode if optimizations are turned on (except for the variable name mangling). Note that in VB .NET the project properties window looks a little different from the C# project properties window, as it relates to optimization configuration. You can turn them on and off by going to the Optimizations node under Configuration Properties and selecting Enable optimizations as Figure 6-2 shows.

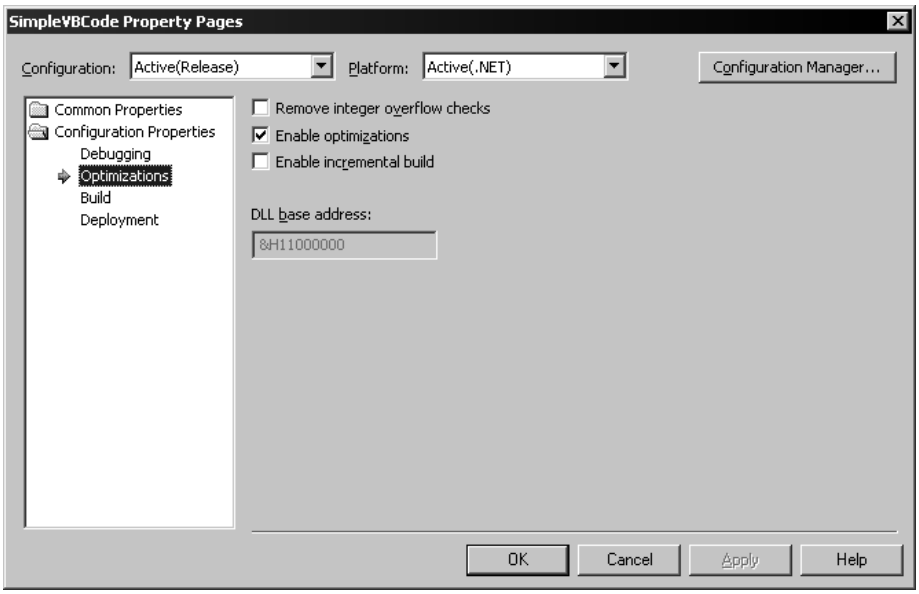

Figure 6-2. Project properties in VB .NET

There are some differences, though, between debug and release builds with optimizations off, as well as how VB .NET implements the code compared to C#. Listing 6-3 shows the CIL code in release mode with no optimizations.

Listing 6-3. VB .NET Compilation Results in Release Mode with Optimizations

```
//  VB .NET Release - no optimizations
.method public instance bool  TestForTypeName() cil managed
{
  // Code size       42 (0x2a)
  .maxstack  3
```

```
  .locals init (class [mscorlib]System.Type V_0,
           bool V_1,
           string V_2,
           int32 V_3,
           bool V_4)
  IL_0000:  ldarg.0
  IL_0001:  callvirt   instance class [mscorlib]System.Type
    [mscorlib]System.Object::GetType()
  IL_0006:  stloc.0
  IL_0007:  ldloc.0
  IL_0008:  callvirt   instance string
    [mscorlib]System.Reflection.MemberInfo::get_Name()
  IL_000d:  stloc.2
  IL_000e:  ldc.i4.1
  IL_000f:  ldloc.2
  IL_0010:  ldstr      "SimpleCode"
  IL_0015:  callvirt   instance bool
    [mscorlib]System.String::Equals(string)
  IL_001a:  bne.un.s   IL_0024
  IL_001c:  ldc.i4.1
  IL_001d:  stloc.s    V_4
  IL_001f:  ldloc.s    V_4
  IL_0021:  stloc.1
  IL_0022:  br.s       IL_0028
  IL_0024:  ldc.i4.0
  IL_0025:  stloc.1
  IL_0026:  br.s       IL_0028
  IL_0028:  ldloc.1
  IL_0029:  ret
} // end of method SimpleCode::TestForTypeName
```

Notice that VB .NET does not create a dummy variable to store the return value; rather, it creates a variable (`V_1`) with the same type as the return type. This lets the VB .NET developer use the method name as the return value. It's more prevalent if you look at the debug build shown in Listing 6-4.

Listing 6-4. VB .NET Compilation Results in Debug Mode

```
.method public instance bool  TestForTypeName() cil managed
{
  // Code size       44 (0x2c)
  .maxstack  3
```

```
  .locals init ([0] class [mscorlib]System.Type someType,
           [1] bool TestForTypeName,
           [2] string typeName,
           [3] int32 i,
           [4] bool yes)
  IL_0000:  nop
  IL_0001:  ldarg.0
  IL_0002:  callvirt   instance class [mscorlib]System.Type
    [mscorlib]System.Object::GetType()
  IL_0007:  stloc.0
  IL_0008:  ldloc.0
  IL_0009:  callvirt   instance string
    [mscorlib]System.Reflection.MemberInfo::get_Name()
  IL_000e:  stloc.2
  IL_000f:  ldc.i4.1
  IL_0010:  ldloc.2
  IL_0011:  ldstr      "SimpleCode"
  IL_0016:  callvirt   instance bool [mscorlib]System.String::Equals(string)
  IL_001b:  bne.un.s   IL_0025
  IL_001d:  ldc.i4.1
  IL_001e:  stloc.s    yes
  IL_0020:  ldloc.s    yes
  IL_0022:  stloc.1
  IL_0023:  br.s       IL_002a
  IL_0025:  nop
  IL_0026:  ldc.i4.0
  IL_0027:  stloc.1
  IL_0028:  br.s       IL_002a
  IL_002a:  ldloc.1
  IL_002b:  ret
} // end of method SimpleCode::TestForTypeName
```

As you can see, the local variable at index position 1 is named `TestForTypeName`. This is the variable that's set if you do something like this in code:

```
TestForTypeName = True
```

You'll also note that VB .NET includes some `nop` opcodes in the CIL stream. The reason it does this is to include all of the VB .NET code it can into the debugging experience. For example, here's the CIL code inlined with the VB .NET code:

```
// Source File 'D:\Personal\APress\Programming in CIL\
// Chapter 6 - dotNET Languages and CIL\SimpleVBCode\SimpleVBCode.vb'
//000009:         Public Function TestForTypeName() As Boolean
```

```
  IL_0000:  nop
//000010:             Dim someType As Type = Me.GetType()
  IL_0001:  ldarg.0
  IL_0002:  callvirt   instance class [mscorlib]System.Type
    [mscorlib]System.Object::GetType()
  IL_0007:  stloc.0
```

Compare that code to the CIL code from C#:

```
// Source File 'D:\Personal\APress\Programming in CIL\
// Chapter 6 - dotNET Languages and CIL\
// SimpleCSharpCode\SimpleCSharpCode.cs'
//000018:  Type someType = this.GetType();
  IL_0000:  ldarg.0
  IL_0001:  call       instance class [mscorlib]System.Type
    [mscorlib]System.Object::GetType()
  IL_0006:  stloc.0
```

If you've ever debugged a program in C#, try setting a breakpoint on the method declaration. Even though VS .NET puts the breakpoint on that line of code, you'll see that the breakpoint is pushed down to the first line of executable code when you start up the debugger. However, in VB .NET, it's different. You can set a breakpoint on the method declaration and it won't move. By putting `nop` opcodes into the CIL stream, VB .NET's compiler can bind these "do-nothing" lines of code to the debug version of the assembly.

The Component Pascal Implementation

The last high-level language implementation I'll show you is Component Pascal (CP):

```
MODULE SimpleCPCode;
    IMPORT System := mscorlib_System, CPmain, Console;

    TYPE SimpleCode* = POINTER TO EXTENSIBLE RECORD
        (System.Object) END;

    PROCEDURE (this : SimpleCode) TestForTypeName*() :
        BOOLEAN, NEW, EXTENSIBLE;
```

```
        VAR someType : System.Type;
            typeName : System.String;
            i : INTEGER;
            yes : BOOLEAN;
    BEGIN
        someType := this.GetType();
        typeName := someType.get_Name();
        IF (typeName.Equals(MKSTR("SimpleCode"))) THEN
            yes := TRUE;
            RETURN yes;
        END;
        RETURN FALSE;
    END TestForTypeName;
END SimpleCPCode.
```

As far as I can tell from the CP docs, there's no debug or release build available, so you'll only be seeing one CIL implementation, which is shown in Listing 6-5.

Listing 6-5. Component Pascal Compilation Results

```
.method public newslot virtual instance bool
        TestForTypeName() cil managed
{
  // Code size       41 (0x29)
  .maxstack  8
  .locals init ([0] class [mscorlib]System.Type someType,
           [1] string typeName,
           [2] int32 i,
           [3] bool yes)
  IL_0000:  ldarg.0
  IL_0001:  call
    instance class [mscorlib]System.Type object::GetType()
  IL_0006:  stloc.0
  IL_0007:  ldloc.0
  IL_0008:  callvirt
    instance string [mscorlib]System.Reflection.MemberInfo::get_Name()
  IL_000d:  stloc.1
  IL_000e:  ldloc.1
  IL_000f:  ldstr      "SimpleCode"
  IL_0014:  call       string [RTS]CP_rts::mkStr(char[])
  IL_0019:  call       instance bool string::Equals(string)
  IL_001e:  brfalse    IL_0027
  IL_0023:  ldc.i4.1
```

```
  IL_0024:  stloc.3
  IL_0025:  ldloc.3
  IL_0026:  ret
  IL_0027:  ldc.i4.0
  IL_0028:  ret
} // end of method SimpleCode::TestForTypeName
```

There really isn't anything interesting with this code except the call to `MKSTR()`, which originates from CP's runtime assembly (RTS). This is necessary because CP needs to resolve the call to `Equals()` since it's overloaded by `System.String`.[3] Because the "SimpleCode" literal could be a `String` or an `Object`, CP's compiler can't resolve the call on its own. Note that I could've made a separate `String` variable to make the call unambiguous:

```
VAR targetName : System.String;
targetName := "SimpleCode";
IF (typeName.Equals(targetName)) THEN
```

Commentary

In all three cases, the implementations are pretty similar, but with some slight differences. C#'s compiler is pretty aggressive in eliminating dead code in comparison to the other two languages.[4] However, there are some optimizations that we as humans can see that the compilers can't. For example, we all know there's no reason to create a `Type` reference to get its name—here's how they could all be optimized:

```
//  C# code
String typeName = this.GetType().Name;
'  VB .NET code
Dim typeName As String = Me.GetType().Name
(*  CP Code *)
typeName := this.GetType().get_Name();
```

3. Later on, I'll demonstrate a more convoluted example with overridden and overloaded methods.

4. For a listing of the optimizations that C#'s compiler will perform, please read the section "Optimizations" in Chapter 36 of Eric Gunnerson's book, *A Programmer's Introduction to C#, Second Edition* (Apress, 2001).

However, it's interesting to see what the compilers do with this optimization. Here's the CIL for all three languages:

```
//  C# and CP CIL
IL_0001:  call  instance class
  [mscorlib]System.Type [mscorlib]System.Object::GetType()
IL_0006:  callvirt instance string
  [mscorlib]System.Reflection.MemberInfo::get_Name()
//  VB .NET CIL
IL_0001:  callvirt instance class
  [mscorlib]System.Type [mscorlib]System.Object::GetType()
IL_0006:  callvirt instance string
  [mscorlib]System.Reflection.MemberInfo::get_Name()
```

Both C# and CP call `GetType()` with `call`, whereas VB .NET calls it with `callvirt`. This is a nonvirtual method, and calling nonvirtual methods with `callvirt` is legal, but why is VB .NET the only language to emit a `callvirt`? It's being more cautious than the other two compilers. `callvirt` will always check to see that the instance reference is the first argument on the stack—if it's `null`, it throws a `NullReferenceException`. `call` won't do this. At the end of the day, they're both legitimate choices. If the reference were `null`, VB .NET would throw the exception before the method is called. In the other two languages, the error wouldn't occur until the reference was accessed (if at all).

SOURCE CODE *The SimpleVBCode, SimpleCSharpCode, and SimpleCPCode projects contain the code written in the different languages discussed in this section.*

Language Constructs

In this section, I'll dive into specific language keywords and constructs and what the generated CIL looks like. It's interesting to see what the compilers are doing with your favorite (or not-so-favorite) language's keywords—in some cases, it might make you question whether you should ever use a certain construct at all.

VB .NET's With Statement

Let's start with an easy one. VB .NET has a `With` statement that allows you to reference an object's members without redundant typing of the object's variable name.

For example, here's a piece of code that prints out the contents of an `Atom` instance:

```
Function PrintAtom(ByVal TargetAtom As Atom)
    Console.WriteLine("Atom name is " & TargetAtom.Name)
    Console.WriteLine("Atom symbol is " & TargetAtom.Symbol)
    Console.WriteLine("Number of protons and electrons:  " & _
        TargetAtom.Electrons)
    Console.WriteLine("Number of neutrons:  " & _
        TargetAtom.Nucleus.Neutrons.Length)
End Function
```

By using `With`, you eliminate the need to type `TargetAtom` whenever you access its property values:

```
Function WithPrintAtom(ByVal TargetAtom As Atom)
    With TargetAtom
        Console.WriteLine("Atom name is " & .Name)
        Console.WriteLine("Atom symbol is " & .Symbol)
        Console.WriteLine("Number of protons and electrons:  " & .Electrons)
        Console.WriteLine("Number of neutrons:  " & .Nucleus.Neutrons.Length)
    End With
End Function
```

Note that `With` only works on objects, so you can't use `With` on `Console` to make the `WriteLine()` calls smaller from a typing perspective.

When you're accessing a number of a particular object's properties and methods, `With` is a nice piece of syntactic sugar to make the code a bit cleaner. In fact, it's so nice that when I wear my C# hat I wish it had a similar construct. The only way to get close to faking it is to set a variable with a very short name (like `x`) equal to the object reference in question:

```
Atom PrintAtom(Atom TargetAtom)
{
    Atom x = TargetAtom;
    Console.WriteLine("Atom name is " + x.Name);
    //  etc.
}
```

What's interesting is that VB .NET is doing the same thing behind the scenes. Let's take a look at some of the CIL produced by these two methods. Here's the first two `WriteLine()` calls in `PrintAtom()`:

```
.method public static object  PrintAtom(
  class WithTest.WithTest/Atom TargetAtom) cil managed
{
  // Code size       106 (0x6a)
  .maxstack  2
  .locals init (object V_0)
  IL_0000:  ldstr      "Atom name is "
  IL_0005:  ldarg.0
  IL_0006:  callvirt   instance string WithTest.WithTest/Atom::get_Name()
  IL_000b:  call       string [mscorlib]System.String::Concat(string,
                                                              string)
  IL_0010:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0015:  ldstr      "Atom symbol is "
  IL_001a:  ldarg.0
  IL_001b:  callvirt   instance string WithTest.WithTest/Atom::get_Symbol()
  IL_0020:  call       string [mscorlib]System.String::Concat(string,
                                                              string)
  IL_0025:  call       void [mscorlib]System.Console::WriteLine(string)
```

As expected, `TargetAtom` is loaded each time information is needed out of it (`ldarg.0`). Now look at the same code in `WithPrintAtom()`:

```
.method public static object  WithPrintAtom
  (class WithTest.WithTest/Atom TargetAtom) cil managed
{
  // Code size       110 (0x6e)
  .maxstack  2
  .locals init (object V_0,
           class WithTest.WithTest/Atom V_1)
  IL_0000:  ldarg.0
  IL_0001:  stloc.1
  IL_0002:  ldstr      "Atom name is "
  IL_0007:  ldloc.1
  IL_0008:  callvirt   instance string WithTest.WithTest/Atom::get_Name()
  IL_000d:  call       string [mscorlib]System.String::Concat(string,
                                                              string)
  IL_0012:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0017:  ldstr      "Atom symbol is "
  IL_001c:  ldloc.1
  IL_001d:  callvirt   instance string WithTest.WithTest/Atom::get_Symbol()
  IL_0022:  call       string [mscorlib]System.String::Concat(string,
                                                              string)
  IL_0027:  call       void [mscorlib]System.Console::WriteLine(string)
```

In this case, the compiler created a local variable of type `Atom` and set it equal to `TargetAtom` (`ldarg.0` and `stloc.1`). Then, whenever `TargetAtom` is needed, VB .NET actually loads the local variable (`ldloc.1`) and not the original variable.

This may seem a bit odd at first glance. Why create the dummy variable? In this case, there's not much of a difference between a local variable and a method argument. However, the reasoning behind this process becomes clearer when you start calling properties or methods as part of the `With` statement. Take a look at the following code:

```
With TargetAtom.Symbol
    Console.WriteLine("Atom object information:  " & .ToString)
End With
```

It's possible that the value returned from `Symbol` would change if you loaded `TargetAtom` onto the stack to get its value to implement the `With` statement. The way `With` works is that the value used when the `With` block is entered doesn't change throughout the block.[5] So the compiler doesn't have much of a choice but to cache the value once and then use that value throughout the lifetime of the `With` block.

SOURCE CODE *The WithTest project contains the methods discussed in this section.*

Implementing Interface Methods

As you have seen, .NET allows you to create interfaces that classes can implement. Furthermore, it's possible to specify which method on which interface your class is implementing.[6] However, the way that this is done varies between languages.

5. See Section 8.4 of the Visual Basic Language Specification of the .NET SDK.
6. This isn't possible with ATL out of the box; see `http://www.sellsbrothers.com/tools/default.aspx` for a workaround to this problem if you run into it and you're still coding COM servers in ATL.

Consider the following set of interface definitions:

```
namespace DriveInterfaces
{
    public interface IGolfer
    {
        string Drive();
    }

    public interface IStockCarRacer
    {
        string Drive();
    }

    public interface ISundayDriver
    {
        string Drive();
    }
}
```

Now, let's say I need to model two different classes: a person who is a stock car racer and a golfer, and another person who's a bad automobile driver as well as a bad golfer. Therefore, the way the first person drives the stock car is different than the way she drives a golf ball; the second person is lousy at both activities. Listing 6-6 shows how this is implemented in C#.

Listing 6-6. Interface Implementation in C#

```
public class StockCarGolfer : IStockCarRacer, IGolfer
{
    public StockCarGolfer() {}

    string IStockCarRacer.Drive()
    {
        return "Without rubbing you ain't got racing!";
    }

    string IGolfer.Drive()
    {
        return "350 yards right down the middle of the fairway...";
    }
}

public class BadDriver : ISundayDriver, IGolfer
{
```

```
    public BadDriver() {}

    string ISundayDriver.Drive()
    {
        return this.Drive();
    }

    string IGolfer.Drive()
    {
        return this.Drive();
    }

    private string Drive()
    {
        return "I'm a bad driver no matter what I do.";
    }
}
```

In C#, interface methods can be overridden by adding methods to the class that match the interface's methods. In this case, you can't add a method called `Drive()` because each interface defines the exact same method. Therefore, C# allows you to define which interface method you're implementing by explicitly stating the method along with the interface name (`string IGolfer.Drive()`, for example). This is known as *explicit interface inheritance*, and the resulting CIL looks like this:

```
.method private hidebysig newslot final virtual
        instance string  DriveInterfaces.IGolfer.Drive() cil managed
{
  .override [DriveInterfaces]DriveInterfaces.IGolfer::Drive
  // Code size       7 (0x7)
  .maxstack  1
  IL_0000:  ldarg.0
  IL_0001:  call       instance string CSharpDrivers.BadDriver::Drive()
  IL_0006:  ret
} // end of method BadDriver::DriveInterfaces.IGolfer.Drive
```

Note that the method name includes the interface name along with the interface's assembly name. The method is `private`, so you can't access the method from outside the class, nor can you access the method from within the class:

```
//  This works
this.Drive();
//  This doesn't
this.DriveInterfaces.IGolfer.Drive();
```

The technique works well if the methods don't share implementations, as is the case with `StockCarGolfer`. However, you'll note that in `BadDriver`, both methods call the same implementation. In C#, there's no way to state that a method implements more than one interface method. With VB .NET, though, you have more flexibility in terms of how you implement the interface's methods. Listing 6-7 shows the implementation of the two classes in VB .NET.

Listing 6-7. Interface Implementation in VB .NET

```
Public Class StockCarGolfer
    Implements IStockCarRacer, IGolfer

    Public Sub New()
    End Sub

    Private Function StockCarDrive() As String _
        Implements IStockCarRacer.Drive

        Return "Without rubbing you ain't got racing!"
    End Function

    Public Function GolfDrive() As String _
        Implements IGolfer.Drive

        Return "350 yards right down the middle of the fairway..."
    End Function
End Class

Public Class BadDriver
    Implements ISundayDriver, IGolfer

    Public Sub New()
    End Sub

    Public Function Drive() As String _
        Implements ISundayDriver.Drive, IGolfer.Drive

        Return "I'm a bad driver no matter what I do."
    End Function
End Class
```

With the `Implements` keyword you can specify which methods will be implemented by the current method. In this case, the implementation of `Drive()` in `BadDriver` looks like this in CIL:

```
.method public newslot final virtual instance string
        Drive() cil managed
{
  .override [DriveInterfaces]DriveInterfaces.ISundayDriver::Drive
  .override [DriveInterfaces]DriveInterfaces.IGolfer::Drive
  // Code size       6 (0x6)
  .maxstack  1
  .locals init (string V_0)
  IL_0000:  ldstr      "I'm a bad driver no matter what I do."
  IL_0005:  ret
} // end of method BadDriver::Drive
```

Note that the method is also declared as `public`, so it's possible to use the method as a client of `BadDriver` as well as from within the type:

```
'  External
Dim bd As BadDriver = new BadDriver
bd.Drive
'  Internal
Me.Drive()
```

Oberon has the same abilities that VB .NET does when it comes to implementing interface methods . . . or so the documentation says. However, the results are quite striking. Listing 6-8 shows the same two types defined in Oberon.

Listing 6-8. Interface Implementation in Oberon

```
MODULE OberonDrivers;
    TYPE StockCarGolfer* = OBJECT
        IMPLEMENTS DriveInterfaces.IStockCarRacer, DriveInterfaces.IGolfer;
        PROCEDURE StockCarDrive() : System.String
            IMPLEMENTS DriveInterfaces.IStockCarRacer.Drive;

        BEGIN
            RETURN "Without rubbing you ain't got racing!";
        END StockCarDrive;
        PROCEDURE GolfDrive*() : System.String
            IMPLEMENTS DriveInterfaces.IGolfer.Drive;
```

```
        BEGIN
            RETURN "350 yards right down the middle of the fairway...";
        END GolfDrive;
    END StockCarGolfer;
    TYPE BadDriver* = OBJECT
        IMPLEMENTS DriveInterfaces.ISundayDriver, DriveInterfaces.IGolfer;
        PROCEDURE SundayDrive() : System.String
            IMPLEMENTS DriveInterfaces.ISundayDriver.Drive;

        BEGIN
            RETURN "I'm a bad driver no matter what I do.";
        END SundayDrive;
        PROCEDURE GolferDrive() : System.String
            IMPLEMENTS DriveInterfaces.IGolfer.Drive;

        BEGIN
            RETURN "I'm a bad driver no matter what I do.";
        END GolferDrive;
    END BadDriver;
END OberonDrivers.
```

You'll be able to compile the code, but if you try to use them in another application, you'll be in for a rude shock. Here's a piece of test code that I created in C# to see what would happen with these types:

```
StockCarGolfer scg = new StockCarGolfer();
IStockCarRacer ISCGStock = scg;
Console.WriteLine("Calling Drive() on StockCarGolfer via IStockCarRacer = " +
    ISCGStock.Drive());
IGolfer ISCGGolf = scg;
Console.WriteLine("Calling Drive() on StockCarGolfer via IGolfer = " +
    ISCGGolf.Drive());
Console.WriteLine("Calling GolfDrive() on StockCarGolfer = " +
    scg.GolfDrive());
Console.WriteLine("Calling StockCarDrive() on StockCarGolfer = " +
    scg.StockCarDrive());
```

I had similar code for `BadDriver`, but it doesn't pay to show it, because the test code won't execute. When I ran this code, I got a `TypeLoadException`. The reason becomes clear when you look at the type's methods in ILDasm:

```
.method public final virtual instance string
        StockCarDrive() cil managed
{
```

```
  // Code size       6 (0x6)
  .maxstack  11
  IL_0000:  ldstr      "Without rubbing you ain't got racing!"
  IL_0005:  ret
} // end of method StockCarGolfer::StockCarDrive
```

Because no `.override` directive is present, the runtime can't determine that you're actually trying to override `Drive()` from `IStockCarDriver`, so it gives up.[7] In fact, PEVerify doesn't like this assembly either:

```
peverify /il /md OberonDrivers.dll

Microsoft (R) .NET Framework PE Verifier  Version 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

[IL]: Error:  [token  0x02000002] Type load failed.
[IL]: Error:  [token  0x02000003] Type load failed.
2 Errors Verifying OberonDrivers.dll
```

I think this is an excellent example of verifying assemblies that you receive from vendors. In this case, if I ran PEVerify on OberonDrivers.dll, I would've seen a problem before I spent the time to create a test harness.[8]

SOURCE CODE *The DriveInterfaces project contains the interface definitions, and the CSharpDrivers, VBDrivers, and OberonDrivers projects contain the implementations of these interfaces.*

On Error Resume Next, or How to Create a Lot of CIL

If you'd ever programmed in VB before .NET, you know that error handling was pretty rudimentary. You had to use the `goto` statement and then you usually jumped to a label:

---

7. The reason this worked with the Oberon example in Chapter 1 is because the interface and class methods matched, so the runtime was able to determine that the interface was implemented correctly.

8. I think that most vendors will do this before they publish their assemblies. However, it doesn't hurt to do a quick check on them before you use them—the time it takes to run PEVerify on an assembly relative to the time it may take to figure out why something isn't working as expected is worth it in my book.

```
Sub GotoErrorHandling(ByVal X As Integer, ByVal Y As Integer)
    On Error GoTo ErrorHandler

    Dim Z As Integer
    Z = X \ Y

    Exit Sub

ErrorHandler:

End Sub
```

This syntax is still preserved in VB .NET, but VB .NET also allows you to handle exceptions via the `Try-Catch-End Try` blocks:

```
Sub NewErrorHandling(ByVal X As Integer, ByVal Y As Integer)
    Try
        Dim Z As Integer
        Z = X \ Y
    Catch e As Exception

    End Try
End Sub
```

Of course, you can simply ignore errors if you want:

```
Sub NoErrorHandling(ByVal X As Integer, ByVal Y As Integer)
    Dim Z As Integer
    Z = X \ Y
End Sub
```

However, if a `DivideByZeroException` occurs, you're sunk, unless somewhere up the call stack a method will catch the exception. To allow code to execute without letting an exception trickle up the stack, you can use `On Error Resume Next`:

```
Sub ResumeNextErrorHandling(ByVal X As Integer, ByVal Y As Integer)
    On Error Resume Next
    Dim Z As Integer
    Z = X \ Y
End Sub
```

Although you may be tempted to use `On Error Resume Next`,[9] I would strongly recommend another approach. For one, you may be suppressing exceptions that simply should not be suppressed. If you're trying to open a file and the runtime won't let you for security reasons, it's far better to catch that exception than it is to have your file processing code continue. The other reason is the code bloat that happens, even with two lines of VB .NET code, as just demonstrated. Let's look at each method in detail to see what's going on. First, take a look at `NoErrorHandling()`:

```
.method public static void  NoErrorHandling(int32 X,
                                             int32 Y) cil managed
{
  // Code size       5 (0x5)
  .maxstack  2
  .locals init (int32 V_0)
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  div
  IL_0003:  stloc.0
  IL_0004:  ret
} // end of method ErrorResumeNext::NoErrorHandling
```

Nothing surprising here—the two arguments are loaded and then `X` is divided by `Y`. Now let's look at the new, modern way of handling exceptions in VB .NET with `NewErrorHandling()`:

```
.method public static void  NewErrorHandling(int32 X,
                                              int32 Y) cil managed
{
  // Code size       21 (0x15)
  .maxstack  2
  .locals init (int32 V_0,
           class [mscorlib]System.Exception V_1)
  .try
  {
    IL_0000:  ldarg.0
    IL_0001:  ldarg.1
    IL_0002:  div
    IL_0003:  stloc.0
    IL_0004:  leave.s    IL_0014
  }  // end .try
  catch [mscorlib]System.Exception
```

9. I have to admit, I've used it on VB projects in the past, primarily in the error handling code itself. But I've made a resolution to never use this feature as long as I live, especially after seeing what happens when it's used!

```
{
    IL_0006:  dup
    IL_0007:  call void [Microsoft.VisualBasic]
      Microsoft.VisualBasic.CompilerServices.ProjectData::SetProjectError(
      class [mscorlib]System.Exception)
    IL_000c:  stloc.1
    IL_000d:  call void [Microsoft.VisualBasic]
      Microsoft.VisualBasic.CompilerServices.ProjectData::ClearProjectError()
    IL_0012:  leave.s    IL_0014
  }  // end handler
  IL_0014:  ret
} // end of method ErrorResumeNext::NewErrorHandling
```

You've seen `.try` blocks in CIL before, but the code that was added to the `catch` block wasn't expected. `SetProjectError()` and `ClearProjectError()` are methods that are used by the VB .NET runtime (Microsoft.VisualBasic.dll) to set the current exception object (`Err`) because you don't have to specify an exception variable in the `catch` statement as I am doing. This is a holdover from pre-.NET VB syntax in which a global exception object was available; in VB .NET, the `Err` object is still available and can be used in the exception handler to catch the (probable) `DivideByZeroException`.

Let's move on to `GotoErrorHandler()`, which is shown in Listing 6-9.

Listing 6-9. "On Error Goto" Results in CIL

```
.method public static void  GotoErrorHandling(int32 X,
                                               int32 Y) cil managed
{
  // Code size       84 (0x54)
  .maxstack  2
  .locals init (int32 V_0,
           class [mscorlib]System.Exception V_1,
           int32 V_2,
           int32 V_3)
  IL_0000:  call       void [Microsoft.VisualBasic]
    Microsoft.VisualBasic.CompilerServices.ProjectData::ClearProjectError()
  IL_0005:  ldc.i4.1
  IL_0006:  stloc.3
  IL_0007:  ldarg.0
  IL_0008:  ldarg.1
  IL_0009:  div
  IL_000a:  stloc.0
  IL_000b:  leave.s    IL_004b
  IL_000d:  leave.s    IL_004b
```

```
  IL_000f:  isinst     [mscorlib]System.Exception
  IL_0014:  brfalse.s  IL_001f
  IL_0016:  ldloc.3
  IL_0017:  brfalse.s  IL_001f
  IL_0019:  ldloc.2
  IL_001a:  brtrue.s   IL_001f
  IL_001c:  ldc.i4.1
  IL_001d:  br.s       IL_0020
  IL_001f:  ldc.i4.0
  IL_0020:  endfilter
  IL_0022:  castclass  [mscorlib]System.Exception
  IL_0027:  dup
  IL_0028:  call       void [Microsoft.VisualBasic]
    Microsoft.VisualBasic.CompilerServices.ProjectData::SetProjectError(
    class [mscorlib]System.Exception)
  IL_002d:  stloc.1
  IL_002e:  ldloc.2
  IL_002f:  brfalse.s  IL_0033
  IL_0031:  leave.s    IL_004b
  IL_0033:  ldc.i4.m1
  IL_0034:  stloc.2
  IL_0035:  ldloc.3
  IL_0036:  switch     (
                        IL_0045,
                        IL_0047)
  IL_0043:  leave.s    IL_0049
  IL_0045:  leave.s    IL_0049
  IL_0047:  leave.s    IL_000d
  IL_0049:  rethrow
  IL_004b:  ldloc.2
  .try IL_0000 to IL_000f filter IL_000f handler IL_0022 to IL_004b
  IL_004c:  brfalse.s  IL_0053
  IL_004e:  call       void [Microsoft.VisualBasic]
    Microsoft.VisualBasic.CompilerServices.ProjectData::ClearProjectError()
  IL_0053:  ret
} // end of method ErrorResumeNext::GotoErrorHandling
```

Basically, VB .NET's compiler adds a `.try` block with a `filter` block to implement the jump to the `ErrorHandler` label when an exception occurs. The exception code that VB .NET emits is pretty interesting to trace. The filter starts at `IL_000f`. If the thrown object is an `Exception` type (which it always should be), then `V_3`'s value is loaded (which is set to 1 at `IL_0006`). This is not false, so the break at `IL_0017` doesn't occur. `V_2`'s value is loaded, which is 0 (it's never changed up to this point since it's been initalized). Because `brtrue` won't cause a break to occur,

a 1 is loaded, and you break to `endfilter`. Now a value of 1 is on the stack, so this causes the handler's code to execute. As before, VB .NET stores the exception information via `SetProjectData()`. When `V_2` is loaded again at `IL_002e`, the following `brfalse` causes the code to jump to `IL_0033`, where `V_2` is set to –1. `V_3` is loaded again, and a `switch` statement occurs. Since `V_3` is still 1, the filter block is left at `IL_0047`, which causes another jump at `IL_000d` to `IL_004b`, where the method finally finished execution. Just seeing this code should motivate you to use the new syntax.

Finally, here's the CIL for `ResumeNextErrorHandling()`, which is shown in Listing 6-10. Because the CIL gets pretty long, I've added the original VB .NET code so you can see where the CIL is added to handle the `On Error Resume Next` construct within the method.

Listing 6-10. "On Error Resume Next" Results in CIL

```
//  Sub ResumeNextErrorHandling(ByVal X As Integer, ByVal Y As Integer)
.method public static void  ResumeNextErrorHandling(int32 X,
    int32 Y) cil managed
{
  // Code size       113 (0x71)
  .maxstack  2
  .locals init (int32 V_0,
           int32 V_1,
           class [mscorlib]System.Exception V_2,
           int32 V_3,
           int32 V_4)
  //  On Error Resume Next
  IL_0000:  call       void [Microsoft.VisualBasic]
    Microsoft.VisualBasic.CompilerServices.ProjectData::ClearProjectError()
  IL_0005:  ldc.i4.1
  IL_0006:  stloc.s    V_4
  //  Dim Z As Integer
  //  Z = X \ Y
  IL_0008:  ldc.i4.1
  IL_0009:  stloc.1
  IL_000a:  ldarg.0
  IL_000b:  ldarg.1
  IL_000c:  div
  IL_000d:  stloc.0
  IL_000e:  leave.s    IL_0068
  IL_0010:  ldloc.3
  IL_0011:  ldc.i4.1
  IL_0012:  add
  IL_0013:  ldc.i4.0
```

```
  IL_0014:  stloc.3
  IL_0015:  switch     (
                        IL_0000,
                        IL_0008,
                        IL_000e)
  IL_0026:  leave.s    IL_0066
  IL_0028:  isinst     [mscorlib]System.Exception
  IL_002d:  brfalse.s  IL_0039
  IL_002f:  ldloc.s    V_4
  IL_0031:  brfalse.s  IL_0039
  IL_0033:  ldloc.3
  IL_0034:  brtrue.s   IL_0039
  IL_0036:  ldc.i4.1
  IL_0037:  br.s       IL_003a
  IL_0039:  ldc.i4.0
  IL_003a:  endfilter
  IL_003c:  castclass  [mscorlib]System.Exception
  IL_0041:  dup
  IL_0042:  call       void [Microsoft.VisualBasic]
    Microsoft.VisualBasic.CompilerServices.ProjectData::SetProjectError(
    class [mscorlib]System.Exception)
  IL_0047:  stloc.2
  IL_0048:  ldloc.3
  IL_0049:  brfalse.s  IL_004d
  IL_004b:  leave.s    IL_0066
  IL_004d:  ldloc.1
  IL_004e:  stloc.3
  IL_004f:  ldloc.s    V_4
  IL_0051:  switch     (
                        IL_0060,
                        IL_0062)
  IL_005e:  leave.s    IL_0064
  IL_0060:  leave.s    IL_0064
  IL_0062:  leave.s    IL_0010
  IL_0064:  rethrow
  IL_0066:  ldloc.2
  .try IL_0000 to IL_0028 filter IL_0028 handler IL_003c to IL_0066
  IL_0067:  throw
  //  End Sub
  IL_0068:  ldloc.3
  IL_0069:  brfalse.s  IL_0070
  IL_006b:  call       void [Microsoft.VisualBasic]
    Microsoft.VisualBasic.CompilerServices.ProjectData::ClearProjectError()
  IL_0070:  ret
} // end of method ErrorResumeNext::ResumeNextErrorHandling
```

Wow, did that bloat the code! If someone insists on using this approach to error handling, just ask him or her if all of this code is really worth it just to prevent an exception from being raised if someone makes the following call:

```
ResumeNextErrorHandling(3, 0)
```

I think not. If your language of choice is VB .NET, I strongly recommend using the new exception handling mechanisms rather than using the old constructs. To me, having a code size of 21 (for `NewErrorHandling()`) is much more appealing that 113 (for `ResumeNextErrorHandling()`).

SOURCE CODE *The ErrorResumeNext project contains the methods described in this section.*

Active Objects

You may know that .NET supports threading—that is, you can create threads that will process work separate from the main thread. In C# and VB .NET, you have to manage threads in native .NET code. Oberon, however, tries to abstract some of the details away with the concept of an *active* object. Let's take a look at how this works via the example shown in Listing 6-11.

Listing 6-11. Creating an Active Object in Oberon

```
MODULE GuidGen;
    TYPE Generator* = OBJECT {ACTIVE}
        CONST waitTime = 1000;

        VAR quit : BOOLEAN;
            curGuid : System.Guid;

        PROCEDURE GetGuid*() : System.Guid;
        BEGIN
            RETURN curGuid;
        END GetGuid;

        PROCEDURE Stop*();
        BEGIN
            quit := TRUE;
        END Stop;
```

```
    BEGIN
        quit := FALSE;
        WHILE ~quit DO
            curGuid := System.Guid.NewGuid();
            System.Threading.Thread.Sleep(waitTime);
        END
    END Generator;
END GuidGen.
```

This assembly will contain a type called `Generator`. However, note that the type is adorned with the `ACTIVE` directive. This attribute tells the compiler to run the code that is contained in the type's `BEGIN...END` block on a separate thread. This code creates a new `Guid` every second. To exit the loop, the client must call `Stop()`.

From the client's perspective, it's oblivious that it spawns a new thread when it creates an instance of `Generator`. However, the way Oberon currently generates the code to handle this thread processing is a bit odd. Figure 6-3 shows what a typical .NET developer will see if he or she looks at `Generator` in VS .NET's Object Browser.

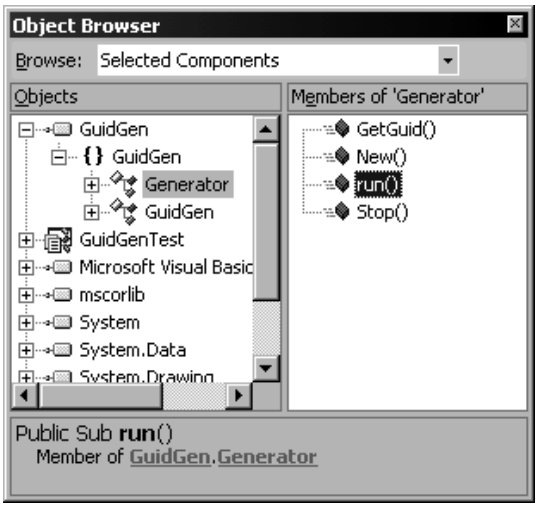

Figure 6-3. Generator's methods

See that `run()` method? This is the method that `Generator`'s constructor passes to a `ThreadStart` delegate to state which method should be run on a separate thread:

```
.method public specialname rtspecialname
        instance void  .ctor() cil managed
{
  // Code size       39 (0x27)
  .maxstack  11
  IL_0000:  ldarg  0
  IL_0004:  call instance void [mscorlib]System.Object::.ctor()
  IL_0009:  ldarg 0
  IL_000d:  ldarg 0
  IL_0011:  ldvirtftn  instance void GuidGen.Generator::run()
  IL_0017:  newobj  instance void
    [mscorlib]System.Threading.ThreadStart::.ctor(object,
    native int)
  IL_001c:  newobj  instance void
    [mscorlib]System.Threading.Thread::.ctor(
    class [mscorlib]System.Threading.ThreadStart)
  IL_0021:  callvirt instance void
    [mscorlib]System.Threading.Thread::Start()
  IL_0026:  ret
} // end of method Generator::.ctor
```

However, there's no reason to make `run()` public, and in this case, it would cause the client a lot of pain if he or she called it on the same thread. `run()` enters a `WHILE...END` loop that will only stop when `quit` is set to `TRUE`. Unless the client happened to call `Stop()` before `run()`, the method would never return, and the client would hang.

Although I think there's a lot of promise for abstracting threading details away from the developer as Oberon does with active objects, I think the compiler designers of Oberon have some work to do before it becomes transparent and seamless. Giving the method that contains the threading code a public scope is dangerous at best. If you use a language construct from any language that you're not completely familiar with, make sure you create a number of test cases before you include it in a larger project. This doesn't guarantee that you will have figured out every possible problem, but you may catch potential issues when the damage caused by these problems is minimal.[10]

10. Right before this book was published, a research paper was released on adding constructs similar to active objects to C#. It's titled "Modern Concurrency Abstractions for C#," and you can download it at `http://research.microsoft.com/Users/luca/Papers/Polyphony%20ECOOP.A4.pdf`.

SOURCE CODE *The GuidGen folder contains the Oberon file to create* `Generator`*, and GuidGenTest contains a VB .NET test harness that allows you to play with* `Generator`.

Language Interoperability: The Real Story

Now that you've gone through investigating the language translations that occur from higher-level languages to CIL via the compilers, let's see what happens when types from different languages are intermixed.

Inheritance with Oberon .NET Types

Let's back up and reexamine the hypothetical coding situation I gave in Chapter 1. I had a number of languages in use to create assemblies that other languages expanded upon. Although the example was pretty basic, you may have been puzzled over one of my design decisions. Recall that I had implemented an interface called `IPerson` (written in C#) in Oberon. This new type was called `Person`. Then, I inherited from `IPerson` to create a new interface called `ICustomer`, which was implemented by a class called `Customer`. All of this was done in C#. Figure 6-4 shows the current design scenario.

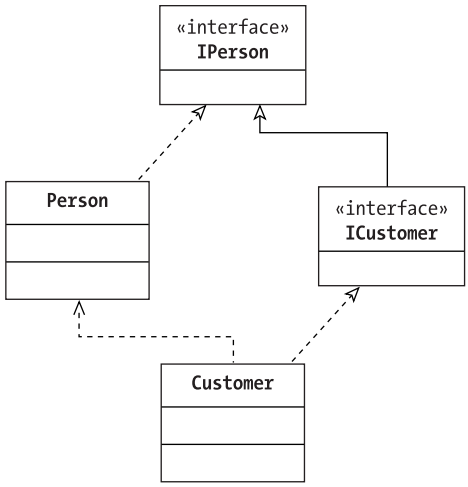

Figure 6-4. Current interface-class relationships

To be honest, I would never have done it this way. Figure 6-5 shows what I would have done if I had the choice.

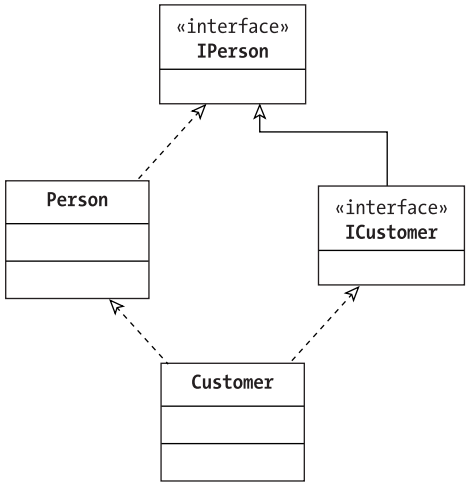

Figure 6-5. Preferred interface-class relationships

The difference is subtle, but it's there. In the first case, `Customer` doesn't inherit from `Person`; it's only using its functionality via containment. In the second case, `Customer` inherits from `Person`.

So what's stopping me from doing this? The issue lies with the Oberon language, or, to be more precise, what Oberon's compiler is doing with the `Person` type. Recall that `Person` is defined as follows:

```
MODULE PersonImpl;
    TYPE Person* = OBJECT IMPLEMENTS PersonDefinition.IPerson;
        (*  code goes here…*)
    END Person;
END PersonImpl.
```

When you load the assembly into ILDasm and look at `Person`, you may be surprised by what you see:

```
.class public auto ansi sealed Person
       extends [mscorlib]System.Object
       implements [PersonDefinition]PersonDefinition.IPerson
{
} // end of class Person
```

The type is sealed! Therefore, there is no way that I can legally extend this type, so that's why I had to use a combination of containment and method forwarding in `Customer` with respect to `Person`.[11]

Although language interoperability is definitely possible in .NET and is much easier than any other solution that I have seen, in some cases, things may not work as expected. I didn't expect Oberon to automatically make any type sealed, so when I tried to make `Customer` inherit from `Person`, I got an error.

Overloaded and Overridden Methods

There are some subtleties in how languages will determine which method they call with respect to types that overload methods from base types. Such subtleties can lead to some interesting discussions in the conference room if you don't look at what's really going on.

Here's a concrete example written in C#:

```
public class Chair {}
public class ComfyChair : Chair {}

public class Person
{
    public string Sit(ComfyChair c)
    {
        MethodBase mb = MethodBase.GetCurrentMethod();
        return "You called " +
            mb.DeclaringType.FullName + "\n" +
            "\t" + mb.ToString() + "\n" +
            "from type " + this.GetType().FullName + "\n";
    }
}

public class SpanishInquisitor : Person
{
    public string Sit(Chair c)
    {
        MethodBase mb = MethodBase.GetCurrentMethod();
        return "You called " +
            mb.DeclaringType.FullName + "\n" +
            "\t" + mb.ToString() + "\n" +
            "from type " + this.GetType().FullName + "\n";
    }
}
```

11. Technically, this is documented behavior according to a white paper from the makers of the Oberon compiler for .NET (`http://www.oberon.ethz.ch/oberon.net/whitepaper/ActiveOberonNetWhitePaper.pdf`). See Section 6.1 in the paper for details.

Both `Sit()` methods are nonvirtual, so `Sit()` in `SpanishInquisitor` is not overriding `Person`'s `Sit()` implementation. Also note that `Sit()` in `SpanishInquisitor` takes a `Chair` instance, but `Sit()` in `Person` takes a more specific type—that is, `ComfyChair`.

Let's create a small test harness of these types in a C# console application:

```
class OAOTest
{
    static void Main(string[] args)
    {
        Chair c = new Chair();
        ComfyChair cc = new ComfyChair();
        Person p = new Person();
        SpanishInquisitor si = new SpanishInquisitor();

        Console.WriteLine(p.Sit(cc));
        Console.WriteLine(si.Sit(cc));
        Console.WriteLine(si.Sit(c));
    }
}
```

Before this code is run, try to guess what the output is going to be. Done? Okay, here's what the console says:

```
You called OverrideAndOverload.Person
        System.String Sit(OverrideAndOverload.ComfyChair)
from type OverrideAndOverload.Person

You called OverrideAndOverload.SpanishInquisitor
        System.String Sit(OverrideAndOverload.Chair)
from type OverrideAndOverload.SpanishInquisitor

You called OverrideAndOverload.SpanishInquisitor
        System.String Sit(OverrideAndOverload.Chair)
from type OverrideAndOverload.SpanishInquisitor
```

The first and third methods aren't really open for discussion, as there are no other choices for the C# compiler to pick. Here's the pertinent CIL that represents the second method call:

```
  .locals init (class [OverrideAndOverload]OverrideAndOverload.Chair V_0,
           class [OverrideAndOverload]OverrideAndOverload.ComfyChair V_1,
           class [OverrideAndOverload]OverrideAndOverload.Person V_2,
           class [OverrideAndOverload]OverrideAndOverload.SpanishInquisitor V_3)
```

```
ldloc.3
ldloc.1
callvirt   instance string [OverrideAndOverload]
  OverrideAndOverload.SpanishInquisitor::Sit(
  class [OverrideAndOverload]OverrideAndOverload.Chair)
```

In this case, C# decides to call `Sit()` on `SpanishInquisitor`. Now let's create a similar test harness in VB .NET:

```
Sub Main()
    Dim c As Chair = New Chair()
    Dim cc As ComfyChair = New ComfyChair()
    Dim p As Person = New Person()
    Dim si As SpanishInquisitor = New SpanishInquisitor()

    Console.WriteLine(p.Sit(cc))
    Console.WriteLine(si.Sit(cc))
    Console.WriteLine(si.Sit(c))
End Sub
```

Looks like the same code, right? But the results are a little different—here's the output:

```
You called OverrideAndOverload.Person
        System.String Sit(OverrideAndOverload.ComfyChair)
from type OverrideAndOverload.Person

You called OverrideAndOverload.Person
        System.String Sit(OverrideAndOverload.ComfyChair)
from type OverrideAndOverload.SpanishInquisitor

You called OverrideAndOverload.SpanishInquisitor
        System.String Sit(OverrideAndOverload.Chair)
from type OverrideAndOverload.SpanishInquisitor
```

And here's the CIL:

```
.locals init (class [OverrideAndOverload]OverrideAndOverload.Chair V_0,
         class [OverrideAndOverload]OverrideAndOverload.ComfyChair V_1,
         class [OverrideAndOverload]OverrideAndOverload.Person V_2,
         class [OverrideAndOverload]OverrideAndOverload.SpanishInquisitor V_3)
ldloc.3
ldloc.1
callvirt   instance string
  [OverrideAndOverload]OverrideAndOverload.Person::Sit(
  class [OverrideAndOverload]OverrideAndOverload.ComfyChair)
```

VB .NET decides to call `Sit()` on `Person` and not on `SpanishInquisitor`. Note that a similar test harness in Oberon yields the same result as the VB .NET test code:

```
MODULE OberonOverrideAndOverloadTest;
VAR
    c: OverrideAndOverload.Chair;
    cc: OverrideAndOverload.ComfyChair;
    p: OverrideAndOverload.Person;
    si: OverrideAndOverload.SpanishInquisitor;
BEGIN
    NEW(c);
    NEW(cc);
    NEW(p);
    NEW(si);
    System.Console.WriteLine{(System.String)}(p.Sit(cc));
    System.Console.WriteLine{(System.String)}(si.Sit(cc));
    System.Console.WriteLine{(System.String)}(si.Sit(c));
END OberonOverrideAndOverloadTest.
```

So what gives? Why do Oberon and VB .NET decide to use the `Sit()` method on `Person`, but C# uses `Sit()` on `SpanishInquisitor`? The reason is that it's purely a language choice—both "sides" make a valid argument. C#'s compiler looks at each object in the inheritance tree, and as soon as it finds a match that's good enough, the compiler calls it. That's why C#'s compiler calls `Sit()` on `SpanishInquisitor`; `ComfyChair` descends from `Chair`, so that call is perfectly valid. Oberon's and VB .NET's compilers look through the tree until they find the best match they can. They therefore call `Sit()` on `Person` because that method signature takes a `ComfyChair` type.

Note that *nothing* in the Partition docs requires a language to take one approach or another.[12] Nor could it—different languages have made different choices in this situation before .NET came along, so it couldn't mandate a rule in this case. If you're on a project where developers insist on using multiple languages and you run into situations where discrepancies arise in behavior, distill the problem down to its essence and consult the Partition docs. You may find other cases where there is no hard-and-fast rule, so keep this technique in mind.

12. See Section 9.2 of Partition I.

SOURCE CODE *The OverrideAndOverload folder contains the class definitions used in this section (Chair, Person, and so on). The OverrideAndOverloadTest, VBOverrideAndOverloadTest, and OberonOverrideAndOverloadTest folders contain the test harnesses.*

The Other Property

In Chapter 2 I said in the section "Defining Properties in Types" that I would show you how a higher-level language would handle extended property information. That is, if a property is defined with the `.other` directive, what does C# do with it, if anything? I'll show you in a rather unexpected way via a COM server written in VB 6.

Let's say you have two classes in a COM server called GetLetSet: `GLS` and `GLSTest`. Here's the definition of `GLS`:

```
Private m_Def As Long

Public Property Let ThisIsTheDefault(ByVal Value As Long)
m_Def = Value
End Property

Public Property Get ThisIsTheDefault() As Long
ThisIsTheDefault = m_Def
End Property
```

`GLS` has one property, `ThisIsTheDefault`. Now, you can't see this from the code, but this property is set as the default property. You can do this in the Procedure Attributes dialog box, which you bring up by selecting Tools ➤ Procedure Attributes. Figure 6-6 shows you where you can make a property the default one for a class.

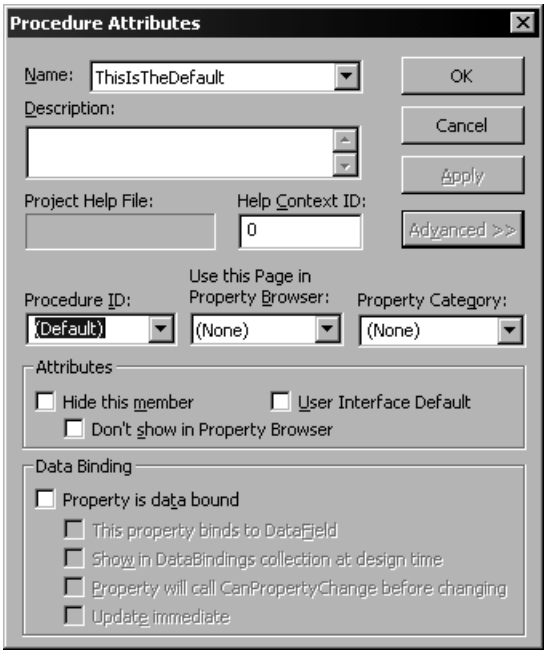

Figure 6-6. Setting the default property in VB 6

When (Default) is selected in the Procedure ID drop-down box, that property will become the default.

If a property is set up as the default property, you don't have to specify the property name to use it. Therefore, the following code in VB 6 is perfectly valid (albeit very confusing):

```
Dim g As GLS
Set g = New GLS
g = 4
```

After this code is complete, the private field `m_Def` will be set to 4. Again, this is not obvious to a developer seeing the code for the first time, which is why few VB 6 developers ever used default properties. And it can be even worse if a developer adds a class like `GLSTest`:

```
Dim m_GLS As GLS

Public Property Let MyGLS(Value As GLS)
m_GLS = Value
End Property
```

```
Public Property Set MyGLS(Value As GLS)
Set m_GLS = Value
End Property

Public Property Get MyGLS() As GLS
Set MyGLS = m_GLS
End Property

Private Sub Class_Initialize()
Set m_GLS = New GLS
End Sub
```

By having a `Let` property defined for `MyGLS`, a VB 6 client can write convoluted code like this:

```
Dim g As GLS
Set g = New GLS
g = 4
Dim gt As GLSTest
Set gt = New GLSTest
gt.MyGLS = g
```

At first glance, it looks like `GLSTest`'s private field `m_GLS` is being set to `g`, when in reality, the last line of code is calling `ThisIsTheDefault` on `m_GLS`, setting `m_Def` equal to 4. This is not what I consider self-documenting code.

However, `get`/`set`/`let` properties lead to an interesting scenario with COM interoperability in .NET. If you reference the GetLetSet.dll COM server in a C# project, you can write C# that does the same thing as the last VB 6 code snippet, but be careful! It's not as straightforward as it may look at first glance:

```
static void Main(string[] args)
{
    GLS gls = new GLSClass();
    gls.ThisIsTheDefault = 4;

    GLSTest gt = new GLSTestClass();
    GLS glRet = gt.get_MyGLS();
    Console.WriteLine("Before:  " +
        glRet.ThisIsTheDefault.ToString());

    gt.let_MyGLS(ref gls);
    glRet = gt.get_MyGLS();
    Console.WriteLine("After:  " +
        glRet.ThisIsTheDefault.ToString());
}
```

The first aspect that's different is that you must explicitly call the property `ThisIsTheDefault` on gls. But the real twister is that you can't use the property `MyGLS`; you have to call the `get_MyGLS()` and `let_MyGLS()` methods. The reason is found when you look at the .NET-to-COM interop assembly that C# creates so you can access `GetLetSet`. This assembly is called Interop.GetLetSet.dll, and you can find it in the bin directories. If you open it up in ILDasm, here's what the `MyGLS` property looks like:

```
.property class GetLetSet.GLS MyGLS()
{
  .custom instance void
    [mscorlib]System.Runtime.InteropServices.DispIdAttribute::.ctor(int32) =
    ( 01 00 00 00 03 68 00 00 )                       // .....h..
  .get instance class
    GetLetSet.GLS GetLetSet._GLSTest::get_MyGLS()
  .set instance void
    GetLetSet._GLSTest::set_MyGLS(class GetLetSet.GLS&)
  .other instance void
    GetLetSet._GLSTest::let_MyGLS(class GetLetSet.GLS&)
} // end of property _GLSTest::MyGLS
```

The `.other` directive is used to represent the `Let` version of `MyGLS`. However, C# gets confused when you try to use `MyGLS`, as it can't figure out if you're really trying to call the `set_MyGLS()` method or the `let_MyGLS()` method. That's why an explicit call to the property's method is necessary.[13]

Admittedly, this is something you probably won't see very often (at least, I hope you won't). But if you ever need to use a COM server where a property allows you to get, let, and set a value, you'll know how to handle it.

SOURCE CODE *The GetLetSet folder contains the VB 6 definitions of GLS and GLSTest. The GLSClient subfolder contains the C# client code.*

13. Unfortunately, IntelliSense won't show the property methods, even though they're public. But if you simply type them in, everything will compile normally.

Overloading Methods in CIL

Let's close out this chapter with a discussion on overloading methods in CIL. Way back in Chapter 2 in the section "Overriding Methods," I gave a definition of a method signature. One of the parts that made up the signature was the return type. I didn't make it explicit in Chapter 2, but now I'm going to show you how you can overload methods in which the return type alone distinguishes the method and how C# and VB .NET each handle these methods.

Let's create a class that has two methods named `GiveMeANumber()`. One returns an `int32`, and the other returns a `float64`:

```
.class public GetNumbers
{
   .method public hidebysig
        specialname rtspecialname
        instance void .ctor() cil managed
    {
       .maxstack 1
       ldarg.0
       call instance void [mscorlib]System.Object::.ctor()
       ret
    }

    .method public instance int32
        GiveMeANumber() cil managed
    {
        .maxstack 1
        ldc.i4 24
        ret
    }

    .method public instance float64
        GiveMeANumber() cil managed
    {
        .maxstack 1
        ldc.r8 42
        ret
    }
}
```

The implementations are pretty easy. The `int32` version will always return 24, and the `float64` version will always return 42.

To call these methods from another assembly written in CIL should be second nature for you by now. As you know, calling methods in CIL requires you to give the type of the return value, so the ilasm compiler will have enough information to discern which `GiveMeANumber()` method you're calling, as demonstrated in Listing 6-12.

Listing 6-12. Resolving Method Calls Based on the Return Value Types

```
.class public OBRTester
{
   .method private hidebysig
        specialname rtspecialname
        instance void .ctor() cil managed
    {
       .maxstack 1
       ldarg.0
       call instance void [mscorlib]System.Object::.ctor()
       ret
    }

    .method public static void Main()
        cil managed
    {
       .entrypoint
       .maxstack 2
       .locals init (class [OverloadByReturn]GetNumbers gn)
        newobj instance void [OverloadByReturn]GetNumbers::.ctor()
        stloc gn
        ldloc gn
        call instance int32 [OverloadByReturn]GetNumbers::GiveMeANumber()
        call void [mscorlib]System.Console::WriteLine(int32)
        ldloc gn
        call instance float64 [OverloadByReturn]GetNumbers::GiveMeANumber()
        call void [mscorlib]System.Console::WriteLine(float64)
        ret
    }
}
```

When the application is run that contains OBRTester, the console should look like this:

```
C:\OBRTester>OBRClient
24
42
```

You get the `int32` first, and then you obtain the `float64` value. At each call opcode, the return type is specified, so all is well in the .NET world.

However, things get pretty ugly in both C# and VB .NET if they encounter `GetNumbers`:

```
//  C#
class OBRTester
{
    static void Main(string[] args)
    {
        GetNumbers gn = new GetNumbers();
        int gnInt = gn.GiveMeANumber();
        Console.WriteLine(gnInt);
        double gnDouble = gn.GiveMeANumber();
        Console.WriteLine(gnDouble);
    }
}

'  VB . NET
Module OBRTester

    Sub Main()
        Dim gn As GetNumbers = New GetNumbers()
        Dim gnInt As Integer = gn.GiveMeANumber()
        Console.WriteLine(gnInt)
        Dim gnDouble As Double = gn.GiveMeANumber()
        Console.WriteLine(gnDouble)
    End Sub

End Module
```

Unfortunately, neither one of these code snippets will compile. When you compile the C# code, the compiler gives you the following error:

```
The call is ambiguous between the following methods or properties:
'GetNumbers.GiveMeANumber()' and 'GetNumbers.GiveMeANumber()'
```

VB .NET's error message is similar (although it's a bit more verbose):

```
Overload resolution failed because no accessible 'GiveMeANumber'
is most specific for these arguments:
'Public Function GiveMeANumber() As Double': Not most specific.
'Public Function GiveMeANumber() As Integer': Not most specific.
```

In both cases, the compiler can't resolve the call you're trying to make. Neither C# nor VB .NET supports overloading methods by return value. As overloading methods based on the return value is not CLS compliant,[14] you should not expose any methods in CIL that do this. Instead, provide a method that can internally resolve which method should be called—this will allow languages like C# and VB .NET to indirectly call these overloaded methods.

SOURCE CODE *The OverloadByReturn folder contains the IL files that define GetNumbers and OBRTester. The CSharpTester and VBTester projects show how you can write code to call* `GetNumbers`*'s methods (although the projects won't compile).*

Conclusion

In this chapter, you looked at a number of different language constructs and interoperability situations and how they really worked at a CIL level. By knowing how CIL works, you could easily determine why compilers make the choices that they do to implement the developer's wishes in a higher-level language. In the next chapter, you're going to start making those choices yourself when you create your own assemblies via the Emitter classes.

14. See Section 9.2 of Partition I.