CHAPTER 8

Methods

In this chapter:

This chapter focuses on the various kinds of methods that you'll run into, including instance constructors and type constructors, as well as how to define methods to overload operators and type conversions (for implicit and explicit casting). We'll also talk about extension methods, which allow you to logically add your own instance methods to already existing types, and partial methods, which allow you to spread a type's implementation into multiple parts.

Instance Constructors and Classes (Reference Types)

Constructors are special methods that allow an instance of a type to be initialized to a good state. Constructor methods are always called `.ctor` (for *constructor*) in a method definition metadata table. When creating an instance of a reference type, memory is allocated for the instance's data fields, the object's overhead fields (type object pointer and sync block index) are initialized, and then the type's instance constructor is called to set the initial state of the object.

When constructing a reference type object, the memory allocated for the object is always zeroed out before the type's instance constructor is called. Any fields that the constructor doesn't explicitly overwrite are guaranteed to have a value of 0 or `null`.

Unlike other methods, instance constructors are never inherited. That is, a class has only the instance constructors that the class itself defines. Because instance constructors are never inherited, you cannot apply the following modifiers to an instance constructor: `virtual`, `new`, `override`, `sealed`, or `abstract`. If you define a class that does not explicitly define any constructors, the C# compiler defines a default (parameterless) constructor for you whose implementation simply calls the base class's parameterless constructor.

For example, if you define the following class.

```
public class SomeType {
}
```

it is as though you wrote the code as follows.

```
public class SomeType {
   public SomeType() : base() { }
}
```

If the class is `abstract`, the compiler-produced default constructor has `protected` accessibility; otherwise, the constructor is given `public` accessibility. If the base class doesn't offer a parameterless constructor, the derived class must explicitly call a base class constructor or the compiler will issue an error. If the class is `static` (`sealed` and `abstract`), the compiler will not emit a default constructor at all into the class definition.

A type can define several instance constructors. Each constructor must have a different signature, and each can have different accessibility. For verifiable code, a class's instance constructor must call its base class's constructor before accessing any of the inherited fields of the base class. The C# compiler will generate a call to the default base class's constructor automatically if the derived class's constructor does not explicitly invoke one of the base class's constructors. Ultimately, `System.Object`'s public, parameterless constructor gets called. This constructor does nothing—it simply returns. This is because `System.Object` defines no instance data fields, and therefore its constructor has nothing to do.

In a few situations, an instance of a type can be created without an instance constructor being called. In particular, calling `Object`'s `MemberwiseClone` method allocates memory, initializes the object's overhead fields, and then copies the source object's bytes to the new object. Also, a constructor is usually not called when deserializing an object with the runtime serializer. The deserialization code allocates memory for the object without calling a constructor by using the `System.Runtime.Serialization.FormatterServices` type's `GetUninitializedObject` or `GetSafeUninitializedObject` methods (as discussed in Chapter 24, "Runtime Serialization").

Important You should not call any virtual methods within a constructor that can affect the object being constructed. The reason is if the virtual method is overridden in the type being instantiated, the derived type's implementation of the overridden method will execute, but all of the fields in the hierarchy have not been fully initialized. Calling a virtual method would therefore result in unpredictable behavior.

C# offers a simple syntax that allows the initialization of fields defined within a reference type when an instance of the type is constructed.

```
internal sealed class SomeType {
   private Int32 m_x = 5;
}
```

When a SomeType object is constructed, its m_x field will be initialized to 5. How does this happen? Well, if you examine the Intermediate Language (IL) for SomeType's constructor method (also called .ctor), you'll see the code shown here.

```
.method public hidebysig specialname rtspecialname
        instance void  .ctor() cil managed
{
  // Code size       14 (0xe)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  ldc.i4.5
  IL_0002:  stfld      int32 SomeType::m_x
  IL_0007:  ldarg.0
  IL_0008:  call       instance void [mscorlib]System.Object::.ctor()
  IL_000d:  ret
} // end of method SomeType::.ctor
```

In this code, you see that SomeType's constructor contains code to store a 5 into m_x and then calls the base class's constructor. In other words, the C# compiler allows the convenient syntax that lets you initialize the instance fields inline and translates this to code in the constructor method to perform the initialization. This means that you should be aware of code explosion, as illustrated by the following class definition.

```
internal sealed class SomeType {
   private Int32  m_x = 5;
   private String m_s = "Hi there";
   private Double m_d = 3.14159;
   private Byte   m_b;

   // Here are some constructors.
   public SomeType()         { ... }
   public SomeType(Int32 x)  { ... }
   public SomeType(String s) { ...; m_d = 10; }
}
```

When the compiler generates code for the three constructor methods, the beginning of each method includes the code to initialize m_x, m_s, and m_d. After this initialization code, the compiler inserts a call to the base class's constructor, and then the compiler appends to the method the code that appears in the constructor methods. For example, the code generated for the constructor that takes a String parameter includes the code to initialize m_x, m_s, and m_d, call the base class's (Object's) constructor, and then overwrite m_d with the value 10. Note that m_b is guaranteed to be initialized to 0 even though no code exists to explicitly initialize it.

Note The compiler initializes any fields by using the convenient syntax before calling a base class's constructor to maintain the impression that these fields always have a value as the source code appearance dictates. The potential problem occurs when a base class's constructor invokes a virtual method that calls back into a method defined by the derived class. If this happens, the fields initialized by using the convenient syntax have been initialized before the virtual method is called.

Because there are three constructors in the preceding class, the compiler generates the code to initialize `m_x`, `m_s`, and `m_d` three times—once per constructor. If you have several initialized instance fields and a lot of overloaded constructor methods, you should consider defining the fields without the initialization, creating a single constructor that performs the common initialization, and having each constructor explicitly call the common initialization constructor. This approach will reduce the size of the generated code. Here is an example using C#'s ability to explicitly have a constructor call another constructor by using the `this` keyword.

```
internal sealed class SomeType {
   // Do not explicitly initialize the fields here.
   private Int32  m_x;
   private String m_s;
   private Double m_d;
   private Byte   m_b;

   // This constructor sets all fields to their default.
   // All of the other constructors explicitly invoke this constructor.
   public SomeType() {
      m_x = 5;
      m_s = "Hi there";
      m_d = 3.14159;
      m_b = 0xff;
   }

   // This constructor sets all fields to their default, then changes m_x.
   public SomeType(Int32 x) : this() {
      m_x = x;
   }

   // This constructor sets all fields to their default, then changes m_s.
   public SomeType(String s) : this() {
      m_s = s;
   }

   // This constructor sets all fields to their default, then changes m_x & m_s.
   public SomeType(Int32 x, String s) : this() {
      m_x = x;
      m_s = s;
   }
}
```

Instance Constructors and Structures (Value Types)

Value type (`struct`) constructors work quite differently from reference type (`class`) constructors. The common language runtime (CLR) always allows the creation of value type instances, and there is no way to prevent a value type from being instantiated. For this reason, value types don't actually even need to have a constructor defined within them, and the C# compiler doesn't emit default parameterless constructors for value types. Examine the following code.

```
internal struct Point {
   public Int32 m_x, m_y;
}
internal sealed class Rectangle {
   public Point m_topLeft, m_bottomRight;
}
```

To construct a `Rectangle`, the `new` operator must be used, and a constructor must be specified. In this case, the default constructor automatically generated by the C# compiler is called. When memory is allocated for the `Rectangle`, the memory includes the two instances of the `Point` value type. For performance reasons, the CLR doesn't attempt to call a constructor for each value type field contained within the reference type. But as I mentioned earlier, the fields of the value types are initialized to `0`/`null`.

The CLR does allow you to define constructors on value types. The only way that these constructors will execute is if you write code to explicitly call one of them, as in `Rectangle`'s constructor, shown here.

```
internal struct Point {
   public Int32 m_x, m_y;

   public Point(Int32 x, Int32 y) {
      m_x = x;
      m_y = y;
   }
}

internal sealed class Rectangle {
   public Point m_topLeft, m_bottomRight;

   public Rectangle() {
      // In C#, new on a value type calls the constructor to
      // initialize the value type's fields.
      m_topLeft     = new Point(1, 2);
      m_bottomRight = new Point(100, 200);
   }
}
```

A value type's instance constructor is executed only when explicitly called. So if `Rectangle`'s constructor didn't initialize its `m_topLeft` and `m_bottomRight` fields by using the `new` operator to call `Point`'s constructor, the `m_x` and `m_y` fields in both `Point` fields would be 0.

In the `Point` value type defined earlier, no default parameterless constructor is defined. However, let's rewrite that code as follows.

```
internal struct Point {
   public Int32 m_x, m_y;

   public Point() {
      m_x = m_y = 5;
   }
}
```

```
internal sealed class Rectangle {
   public Point m_topLeft, m_bottomRight;

   public Rectangle() {
   }
}
```

Now when a new `Rectangle` is constructed, what do you think the `m_x` and `m_y` fields in the two `Point` fields, `m_topLeft` and `m_bottomRight`, would be initialized to: 0 or 5? (Hint: This is a trick question.)

Many developers (especially those with a C++ background) would expect the C# compiler to emit code in `Rectangle`'s constructor that automatically calls `Point`'s default parameterless constructor for the `Rectangle`'s two fields. However, to improve the run-time performance of the application, the C# compiler doesn't automatically emit this code. In fact, many compilers will never emit code to call a value type's default constructor automatically, even if the value type offers a parameterless constructor. To have a value type's parameterless constructor execute, the developer must add explicit code to call a value type's constructor.

Based on the information in the preceding paragraph, you should expect the `m_x` and `m_y` fields in `Rectangle`'s two `Point` fields to be initialized to 0 in the code shown earlier because there are no explicit calls to `Point`'s constructor anywhere in the code.

However, I did say that my original question was a trick question. The trick part is that C# doesn't allow a value type to define a parameterless constructor. So the previous code won't actually compile. The C# compiler produces the following message when attempting to compile that code: `error CS0568: Structs cannot contain explicit parameterless constructors.`

C# purposely disallows value types from defining parameterless constructors to remove any confusion a developer might have about when that constructor gets called. If the constructor can't be defined, the compiler can never generate code to call it automatically. Without a parameterless constructor, a value type's fields are always initialized to `0`/`null`.

Note Strictly speaking, value type fields are guaranteed to be `0`/`null` when the value type is a field nested within a reference type. However, stack-based value type fields are not guaranteed to be `0`/`null`. For verifiability, any stack-based value type field must be written to prior to being read. If code could read a value type's field prior to writing to the field, a security breach is possible. C# and other compilers that produce verifiable code ensure that all stack-based value types have their fields zeroed out or at least written to before being read so that a verification exception won't be thrown at run time. For the most part, this means that you can assume that your value types have their fields initialized to `0`, and you can completely ignore everything in this note.

Keep in mind that although C# doesn't allow value types with parameterless constructors, the CLR does. So if the unobvious behavior described earlier doesn't bother you, you can use another programming language (such as IL assembly language) to define your value type with a parameterless constructor.

Because C# doesn't allow value types with parameterless constructors, compiling the following type produces the following message: `error CS0573: 'SomeValType.m_x': cannot have instance field initializers in structs.`

```
internal struct SomeValType {
   // You cannot do inline instance field initialization in a value type.
   private Int32 m_x = 5;
}
```

In addition, because verifiable code requires that every field of a value type be written to prior to any field being read, any constructors that you do have for a value type must initialize all of the type's fields. The following type defines a constructor for the value type but fails to initialize all of the fields.

```
internal struct SomeValType {
   private Int32 m_x, m_y;

   // C# allows value types to have constructors that take parameters.
   public SomeValType(Int32 x) {
      m_x = x;
      // Notice that m_y is not initialized here.
   }
}
```

When compiling this type, the C# compiler produces the following message: `error CS0171: Field 'SomeValType.m_y' must be fully assigned before control leaves the constructor.` To fix the problem, assign a value (usually 0) to y in the constructor.

As an alternative way to initialize all the fields of a value type, you can actually do the following.

```
// C# allows value types to have constructors that take parameters.
public SomeValType(Int32 x) {
   // Looks strange but compiles fine and initializes all fields to 0/null.
   this = new SomeValType();

   m_x = x; // Overwrite m_x's 0 with x
   // Notice that m_y was initialized to 0.
}
```

In a value type's constructor, `this` represents an instance of the value type itself and you can actually assign to it the result of newing up an instance of the value type, which really just zeroes out all the fields. In a reference type's constructor, `this` is considered read-only, so you cannot assign to it at all.

Type Constructors

In addition to instance constructors, the CLR supports type constructors (also known as *static constructors*, *class constructors*, or *type initializers*). A type constructor can be applied to interfaces (although C# doesn't allow this), reference types, and value types. Just as instance constructors are used to set the initial state of an instance of a type, type constructors are used to set the initial state of a type. By default, types don't have a type constructor defined within them. If a type has a type

constructor, it can have no more than one. In addition, type constructors never have parameters. In C#, here's how to define a reference type and a value type that have type constructors.

```
internal sealed class SomeRefType {
   static SomeRefType() {
      // This executes the first time a SomeRefType is accessed.
   }
}

internal struct SomeValType {
   // C# does allow value types to define parameterless type constructors.
   static SomeValType() {
      // This executes the first time a SomeValType is accessed.
   }
}
```

You'll notice that you define type constructors just as you would parameterless instance constructors, except that you must mark them as `static`. Also, type constructors should always be private; C# makes them `private` for you automatically. In fact, if you explicitly mark a type constructor as `private` (or anything else) in your source code, the C# compiler issues the following error: `error CS0515: 'SomeValType.SomeValType()': access modifiers are not allowed on static constructors.` Type constructors should be private to prevent any developer-written code from calling them; the CLR is always capable of calling a type constructor.

Important Although you can define a type constructor within a value type, you should never actually do this because there are times when the CLR will not call a value type's static type constructor. Here is an example.

```
internal struct SomeValType {
   static SomeValType() {
      Console.WriteLine("This never gets displayed");
   }
   public Int32 m_x;
}

public sealed class Program {
   public static void Main() {
      SomeValType[] a = new SomeValType[10];
      a[0].m_x = 123;
      Console.WriteLine(a[0].m_x);   // Displays 123
   }
}
```

The calling of a type constructor is a tricky thing. When the just-in-time (JIT) compiler is compiling a method, it sees what types are referenced in the code. If any of the types define a type constructor, the JIT compiler checks if the type's type constructor has already been executed for this AppDomain. If the constructor has never executed, the JIT compiler emits a call to the type constructor into the native code that the JIT compiler is emitting. If the type constructor for the type has already executed, the JIT compiler does not emit the call because it knows that the type is already initialized.

Now, after the method has been JIT-compiled, the thread starts to execute it and will eventually get to the code that calls the type constructor. In fact, it is possible that multiple threads will be executing the same method concurrently. The CLR wants to ensure that a type's constructor executes only once per AppDomain. To guarantee this, when a type constructor is called, the calling thread acquires a mutually exclusive thread synchronization lock. So if multiple threads attempt to simultaneously call a type's static constructor, only one thread will acquire the lock and the other threads will block. The first thread will execute the code in the static constructor. After the first thread leaves the constructor, the waiting threads will wake up and will see that the constructor's code has already been executed. These threads will not execute the code again; they will simply return from the constructor method. In addition, if any of these methods ever get called again, the CLR knows that the type constructor has already executed and will ensure that the constructor is not called again.

Note Because the CLR guarantees that a type constructor executes only once per AppDomain and is thread-safe, a type constructor is a great place to initialize any singleton objects required by the type.

Within a single thread, there is a potential problem that can occur if two type constructors contain code that reference each other. For example, ClassA has a type constructor containing code that references ClassB, and ClassB has a type constructor containing code that references ClassA. In this situation, the CLR still guarantees that each type constructor's code executes only once; however, it cannot guarantee that ClassA's type constructor code has run to completion before executing ClassB's type constructor. You should certainly try to avoid writing code that sets up this scenario. In fact, because the CLR is responsible for calling type constructors, you should always avoid writing any code that requires type constructors to be called in a specific order.

Finally, if a type constructor throws an unhandled exception, the CLR considers the type to be unusable. Attempting to access any fields or methods of the type will cause a `System.TypeInitializationException` to be thrown.

The code in a type constructor has access only to a type's static fields, and its usual purpose is to initialize those fields. As it does with instance fields, C# offers a simple syntax that allows you to initialize a type's static fields.

```
internal sealed class SomeType {
   private static Int32 s_x = 5;
}
```

Note Although C# doesn't allow a value type to use inline field initialization syntax for instance fields, it does allow you to use it for static fields. In other words, if you change the `SomeType` type above from a `class` to a `struct`, the code will compile and work as expected.

When this code is built, the compiler automatically generates a type constructor for SomeType. It's as if the source code had originally been written as follows.

```
internal sealed class SomeType {
   private static Int32 s_x;
   static SomeType() { s_x = 5; }
}
```

Using ILDasm.exe, it's easy to verify what the compiler actually produced by examining the IL for the type constructor. Type constructor methods are always called `.cctor` (for *class constructor*) in a method definition metadata table.

In the code below, you see that the `.cctor` method is `private` and `static`. In addition, notice that the code in the method does in fact load a 5 into the static field `s_x`.

```
.method private hidebysig specialname rtspecialname static
        void  .cctor() cil managed
{
  // Code size       7 (0x7)
  .maxstack  8
  IL_0000:  ldc.i4.5
  IL_0001:  stsfld     int32 SomeType::s_x
  IL_0006:  ret
} // end of method SomeType::.cctor
```

Type constructors shouldn't call a base type's type constructor. Such a call isn't necessary because none of a type's static fields are shared or inherited from its base type.

Note Some languages, such as Java, expect that accessing a type causes its type constructor and all of its base type's type constructors to be called. In addition, interfaces implemented by the types must also have their type constructors called. The CLR doesn't offer this behavior. However, the CLR does offer compilers and developers the ability to provide this behavior via the `RunClassConstructor` method offered by the `System.Runtime.CompilerServices.RuntimeHelpers` type. Any language that requires this behavior would have its compiler emit code into a type's type constructor that calls this method for all base types. When using the `RunClassConstructor` method to call a type constructor, the CLR knows if the type constructor has executed previously and, if it has, the CLR won't call it again.

Finally, assume that you have this code.

```
internal sealed class SomeType {
   private static Int32 s_x = 5;

   static SomeType() {
      s_x = 10;
   }
}
```

In this case, the C# compiler generates a single type constructor method. This constructor first initializes s_x to 5 and then initializes s_x to 10. In other words, when the C# compiler generates IL code for the type constructor, it first emits the code required to initialize the static fields followed by the explicit code contained in your type constructor method.

Important Developers occasionally ask me if there's a way to get some code to execute when a type is unloaded. You should first know that types are unloaded only when the AppDomain unloads. When the AppDomain unloads, the object that identifies the type becomes unreachable, and the garbage collector reclaims the type object's memory. This behavior leads many developers to believe that they could add a static `Finalize` method to the type, which will automatically get called when the type is unloaded. Unfortunately, the CLR doesn't support static `Finalize` methods. All is not lost, however. If you want some code to execute when an AppDomain unloads, you can register a callback method with the `System.AppDomain` type's `DomainUnload` event.

Operator Overload Methods

Some programming languages allow a type to define how operators should manipulate instances of the type. For example, a lot of types (such as `System.String`, `System.Decimal`, and `System.DateTime`) overload the equality (==) and inequality (!=) operators. The CLR doesn't know anything about operator overloading because it doesn't even know what an operator is. Your programming language defines what each operator symbol means and what code should be generated when these special symbols appear.

For example, in C#, applying the + symbol to primitive numbers causes the compiler to generate code that adds the two numbers together. When the + symbol is applied to `String` objects, the C# compiler generates code that concatenates the two strings together. For inequality, C# uses the != symbol, while Microsoft Visual Basic uses the <> symbol. Finally, the ^ symbol means exclusive OR (XOR) in C#, but it means exponent in Visual Basic.

Although the CLR doesn't know anything about operators, it does specify how languages should expose operator overloads so that they can be readily consumed by code written in a different programming language. Each programming language gets to decide for itself whether it will support operator overloads, and if it does, the syntax for expressing and using them. As far as the CLR is concerned, operator overloads are simply methods.

Your choice of programming language determines whether or not you get the support of operator overloading and what the syntax looks like. When you compile your source code, the compiler produces a method that identifies the behavior of the operator. The CLR specification mandates that operator overload methods be `public` and `static` methods. In addition, C# (and many other languages) requires that at least one of the operator method's parameters must be the same as the type

that the operator method is defined within. The reason for this restriction is that it enables the C# compiler to search for a possible operator method to bind to in a reasonable amount of time.

Here is an example of an operator overload method defined in a C# class definition.

```
public sealed class Complex {
   public static Complex operator+(Complex c1, Complex c2) { ... }
}
```

The compiler emits a metadata method definition entry for a method called `op_Addition`; the method definition entry also has the `specialname` flag set, indicating that this is a "special" method. When language compilers (including the C# compiler) see a + operator specified in source code, they look to see if one of the operand's types defines a `specialname` method called `op_Addition` whose parameters are compatible with the operand's types. If this method exists, the compiler emits code to call this method. If no such method exists, a compilation error occurs.

Tables 8-1 and 8-2 show the set of unary and binary operators that C# supports being overloaded, their symbols, and the corresponding Common Language Specification (CLS) method name that the compiler emits. I'll explain the tables' third columns in the next section.

TABLE 8-1 C# Unary Operators and Their CLS-Compliant Method Names

C# Operator Symbol	Special Method Name	Suggested CLS-Compliant Method Name
+	op_UnaryPlus	Plus
-	op_UnaryNegation	Negate
!	op_LogicalNot	Not
~	op_OnesComplement	OnesComplement
++	op_Increment	Increment
--	op_Decrement	Decrement
(none)	op_True	IsTrue { get; }
(none)	op_False	IsFalse { get; }

TABLE 8-2 C# Binary Operators and Their CLS-Compliant Method Names

C# Operator Symbol	Special Method Name	Suggested CLS-Compliant Method Name
+	op_Addition	Add
-	op_Subtraction	Subtract
*	op_Multiply	Multiply
/	op_Division	Divide
%	op_Modulus	Mod
&	op_BitwiseAnd	BitwiseAnd
|	op_BitwiseOr	BitwiseOr
^	op_ExclusiveOr	Xor

C# Operator Symbol	Special Method Name	Suggested CLS-Compliant Method Name
<<	op_LeftShift	LeftShift
>>	op_RightShift	RightShift
==	op_Equality	Equals
!=	op_Inequality	Equals
<	op_LessThan	Compare
>	op_GreaterThan	Compare
<=	op_LessThanOrEqual	Compare
>=	op_GreaterThanOrEqual	Compare

The CLR specification defines many additional operators that can be overloaded, but C# does not support these additional operators. Therefore, they are not in mainstream use, so I will not list them here. If you are interested in the complete list, please see the ECMA specifications (*www.ecma-international.org/publications/standards/Ecma-335.htm*) for the Common Language Infrastructure (CLI), Partition I, Concepts and Architecture, Sections 10.3.1 (unary operators) and 10.3.2 (binary operators).

Note If you examine the core numeric types (`Int32`, `Int64`, `UInt32`, and so on) in the Framework Class Library (FCL), you'll see that they don't define any operator overload methods. The reason they don't is that compilers look specifically for operations on these primitive types and emit IL instructions that directly manipulate instances of these types. If the types were to offer methods and if compilers were to emit code to call these methods, a run-time performance cost would be associated with the method call. Plus, the method would ultimately have to execute some IL instructions to perform the expected operation anyway. This is the reason why the core FCL types don't define any operator overload methods. Here's what this means to you: if the programming language you're using doesn't support one of the core FCL types, you won't be able to perform any operations on instances of that type.

Operators and Programming Language Interoperability

Operator overloading can be a very useful tool, allowing developers to express their thoughts with succinct code. However, not all programming languages support operator overloading. When using a language that doesn't support operator overloading, the language will not know how to interpret the + operator (unless the type is a primitive in that language), and the compiler will emit an error. When using languages that do not support operator overloading, the language should allow you to call the desired `op_*` method directly (such as `op_Addition`).

If you are using a language that doesn't support + operator overloading to be defined in a type, obviously, this type could still offer an `op_Addition` method. From C#, you might expect that you

could call this op_Addition method by using the + operator, but you cannot. When the C# compiler detects the + operator, it looks for an op_Addition method that has the specialname metadata flag associated with it so that the compiler knows for sure that the op_Addition method is intended to be an operator overload method. Because the op_Addition method is produced by a language that doesn't support operator overloads, the method won't have the specialname flag associated with it, and the C# compiler will produce a compilation error. Of course, code in any language can explicitly call a method that just happens to be named op_Addition, but the compilers won't translate a usage of the + symbol to call this method.

Jeff's Opinion About Microsoft's Operator Method Name Rules

I'm sure that all of these rules about when you can and can't call an operator overload method seem very confusing and overly complicated. If compilers that supported operator overloading just didn't emit the specialname metadata flag, the rules would be a lot simpler, and programmers would have an easier time working with types that offer operator overload methods. Languages that support operator overloading would support the operator symbol syntax, and all languages would support calling the various op_ methods explicitly. I can't come up with any reason why Microsoft made this so difficult, and I hope that they'll loosen these rules in future versions of their compilers.

For a type that defines operator overload methods, Microsoft recommends that the type also define friendlier public static methods that call the operator overload methods internally. For example, a public-friendly named method called Add should be defined by a type that overloads the op_Addition method. The third column in Tables 8-1 and 8-2 lists the recommended friendly name for each operator. So the Complex type shown earlier should be defined in the following way.

```
public sealed class Complex {
   public static Complex operator+(Complex c1, Complex c2) { ... }
   public static Complex Add(Complex c1, Complex c2) { return(c1 + c2); }
}
```

Certainly, code written in any programming language can call any of the friendly operator methods, such as Add. Microsoft's guideline that types offer these friendly method names complicates the story even more. I feel that this additional complication is unnecessary, and that calling these friendly named methods would cause an additional performance hit unless the JIT compiler is able to inline the code in the friendly named method. Inlining the code would cause the JIT compiler to optimize the code, removing the additional method call and boosting runtime performance.

Note For an example of a type that overloads operators and uses the friendly method names as per Microsoft's design guidelines, see the System.Decimal class in the FCL.

Conversion Operator Methods

Occasionally, you need to convert an object from one type to an object of a different type. For example, I'm sure you've had to convert a `Byte` to an `Int32` at some point in your life. When the source type and the target type are a compiler's primitive types, the compiler knows how to emit the necessary code to convert the object.

If the source type or target type is not a primitive, the compiler emits code that has the CLR perform the conversion (cast). In this case, the CLR just checks if the source object's type is the same type as the target type (or derived from the target type). However, it is sometimes natural to want to convert an object of one type to a completely different type. For example, the `System.Xml.Linq.XElement` class allows you to convert an Extensible Markup Language (XML) element to a `Boolean`, `(U)Int32`, `(U)Int64`, `Single`, `Double`, `Decimal`, `String`, `DateTime`, `DateTimeOffset`, `TimeSpan`, `Guid`, or the nullable equivalent of any of these types (except `String`). You could also imagine that the FCL included a `Rational` data type and that it might be convenient to convert an `Int32` object or a `Single` object to a `Rational` object. Moreover, it also might be nice to convert a `Rational` object to an `Int32` or a `Single` object.

To make these conversions, the `Rational` type should define public constructors that take a single parameter: an instance of the type that you're converting from. You should also define public instance `ToXxx` methods that take no parameters (just like the very popular `ToString` method). Each method will convert an instance of the defining type to the `Xxx` type. Here's how to correctly define conversion constructors and methods for a `Rational` type.

```
public sealed class Rational {
   // Constructs a Rational from an Int32
   public Rational(Int32 num) { ... }

   // Constructs a Rational from a Single
   public Rational(Single num) { ... }

   // Converts a Rational to an Int32
   public Int32 ToInt32() { ... }

   // Converts a Rational to a Single
   public Single ToSingle() { ... }
}
```

By invoking these constructors and methods, a developer using any programming language can convert an `Int32` or a `Single` object to a `Rational` object and convert a `Rational` object to an `Int32` or a `Single` object. The ability to do these conversions can be quite handy, and when designing a type, you should seriously consider what conversion constructors and methods make sense for your type.

In the previous section, I discussed how some programming languages offer operator overloading. Well, some programming languages (such as C#) also offer conversion operator overloading. *Conversion operators* are methods that convert an object from one type to another type. You define a conversion operator method by using special syntax. The CLR specification mandates that conversion

overload methods be `public` and `static` methods. In addition, C# (and many other languages) requires that either the parameter or the return type must be the same as the type that the conversion method is defined within. The reason for this restriction is that it enables the C# compiler to search for a possible operator method to bind to in a reasonable amount of time. The following code adds four conversion operator methods to the `Rational` type.

```
public sealed class Rational {
   // Constructs a Rational from an Int32
   public Rational(Int32 num) { ... }

   // Constructs a Rational from a Single
   public Rational(Single num) { ... }

   // Converts a Rational to an Int32
   public Int32 ToInt32() { ... }

   // Converts a Rational to a Single
   public Single ToSingle() { ... }

   // Implicitly constructs and returns a Rational from an Int32
   public static implicit operator Rational(Int32 num) {
      return new Rational(num);
   }

   // Implicitly constructs and returns a Rational from a Single
   public static implicit operator Rational(Single num) {
      return new Rational(num);
   }

   // Explicitly returns an Int32 from a Rational
   public static explicit operator Int32(Rational r) {
      return r.ToInt32();
   }

   // Explicitly returns a Single from a Rational
   public static explicit operator Single(Rational r) {
      return r.ToSingle();
   }
}
```

For conversion operator methods, you must indicate whether a compiler can emit code to call a conversion operator method implicitly or whether the source code must explicitly indicate when the compiler is to emit code to call a conversion operator method. In C#, you use the `implicit` keyword to indicate to the compiler that an explicit cast doesn't have to appear in the source code in order to emit code that calls the method. The `explicit` keyword allows the compiler to call the method only when an explicit cast exists in the source code.

After the `implicit` or `explicit` keyword, you tell the compiler that the method is a conversion operator by specifying the `operator` keyword. After the `operator` keyword, you specify the type that an object is being cast to; in the parentheses, you specify the type that an object is being cast from.

Defining the conversion operators in the preceding `Rational` type allows you to write code like this (in C#).

```
public sealed class Program {
   public static void Main() {
      Rational r1 = 5;         // Implicit cast from Int32  to Rational
      Rational r2 = 2.5F;      // Implicit cast from Single to Rational

      Int32  x = (Int32)  r1;  // Explicit cast from Rational to Int32
      Single s = (Single) r2;  // Explicit cast from Rational to Single
   }
}
```

Under the covers, the C# compiler detects the casts (type conversions) in the code and internally generates IL code that calls the conversion operator methods defined by the `Rational` type. But what are the names of these methods? Well, compiling the `Rational` type and examining its metadata shows that the compiler produces one method for each conversion operator defined. For the `Rational` type, the metadata for the four conversion operator methods looks like this.

```
public static Rational op_Implicit(Int32 num)
public static Rational op_Implicit(Single num)
public static Int32    op_Explicit(Rational r)
public static Single   op_Explicit(Rational r)
```

As you can see, methods that convert an object from one type to another are always named `op_Implicit` or `op_Explicit`. You should define an implicit conversion operator only when precision or magnitude isn't lost during a conversion, such as when converting an `Int32` to a `Rational`. However, you should define an explicit conversion operator if precision or magnitude is lost during the conversion, as when converting a `Rational` object to an `Int32`. If an explicit conversion fails, you should indicate this by having your explicit conversion operator method throw an `OverflowException` or an `InvalidOperationException`.

Note The two `op_Explicit` methods take the same parameter, a `Rational`. However, the methods differ by their return type, an `Int32` and a `Single`. This is an example of two methods that differ only by their return type. The CLR fully supports the ability for a type to define multiple methods that differ only by return type. However, very few languages expose this ability. As you're probably aware, C++, C#, Visual Basic, and Java are all examples of languages that don't support the definition of multiple methods that differ only by their return type. A few languages (such as IL assembly language) allow the developer to explicitly select which of these methods to call. Of course, IL assembly language programmers shouldn't take advantage of this ability because the methods they define can't be callable from other programming languages. Even though C# doesn't expose this ability to the C# programmer, the compiler does take advantage of this ability internally when a type defines conversion operator methods.

C# has full support for conversion operators. When it detects code where you're using an object of one type and an object of a different type is expected, the compiler searches for an implicit conversion operator method capable of performing the conversion and generates code to call that method. If an implicit conversion operator method exists, the compiler emits a call to it in the resulting IL code. If the compiler sees source code that is explicitly casting an object from one type to another type, the compiler searches for an implicit or explicit conversion operator method. If one exists, the compiler emits the call to the method. If the compiler can't find an appropriate conversion operator method, it issues an error and doesn't compile the code.

Note C# generates code to invoke explicit conversion operators when using a cast expression; they are never invoked when using C#'s `as` or `is` operators.

To really understand operator overload methods and conversion operator methods, I strongly encourage you to examine the `System.Decimal` type as a role model. `Decimal` defines several constructors that allow you to convert objects from various types to a `Decimal`. It also offers several `ToXxx` methods that let you convert a `Decimal` object to another type. Finally, the type defines several conversion operators and operator overload methods as well.

Extension Methods

The best way to understand C#'s *extension methods* feature is by way of an example. In the "`StringBuilder` Members" section in Chapter 14, "Chars, Strings, and Working with Text," I mention how the `StringBuilder` class offers fewer methods than the `String` class for manipulating a string and how strange this is, considering that the `StringBuilder` class is the preferred way of manipulating a string because it is mutable. So, let's say that you would like to define some of these missing methods yourself to operate on a `StringBuilder`. For example, you might want to define your own `IndexOf` method as follows.

```
public static class StringBuilderExtensions {
   public static Int32 IndexOf(StringBuilder sb, Char value) {
      for (Int32 index = 0; index < sb.Length; index++)
         if (sb[index] == value) return index;
      return -1;
   }
}
```

Now that you have defined this method, you can use it as the following code demonstrates.

```
StringBuilder sb = new StringBuilder("Hello. My name is Jeff.");   // The initial string

// Change period to exclamation and get # characters in 1st sentence (5).
Int32 index = StringBuilderExtensions.IndexOf(sb.Replace('.', '!'), '!');
```

This code works just fine, but is it not ideal from a programmer's perspective. The first problem is that a programmer who wants to get the index of a character within a `StringBuilder` must know

that the StringBuilderExtensions class even exists. The second problem is that the code does not reflect the order of operations that are being performed on the StringBuilder object, making the code difficult to write, read, and maintain. The programmer wants to call Replace first and then call IndexOf; but when you read the last line of code from left to right, IndexOf appears first on the line and Replace appears second. Of course, you could alleviate this problem and make the code's behavior more understandable by rewriting it like this.

```
// First, change period to exclamation mark
sb.Replace('.', '!');

// Now, get # characters in 1st sentence (5)
Int32 index = StringBuilderExtensions.IndexOf(sb, '!');
```

However, a third problem exists with both versions of this code that affects understanding the code's behavior. The use of StringBuilderExtensions is overpowering and detracts a programmer's mind from the operation that is being performed: IndexOf. If the StringBuilder class had defined its own IndexOf method, then we could rewrite the code above as follows.

```
// Change period to exclamation and get # characters in 1st sentence (5).
Int32 index = sb.Replace('.', '!').IndexOf('!');
```

Wow, look how great this is in terms of code maintainability! In the StringBuilder object, we're going to replace a period with an exclamation mark and then find the index of the exclamation mark.

Now, I can explain what C#'s extension methods feature does. It allows you to define a static method that you can invoke using instance method syntax. Or, in other words, we can now define our own IndexOf method and the three problems mentioned above go away. To turn the IndexOf method into an extension method, we simply add the this keyword before the first argument.

```
public static class StringBuilderExtensions {
   public static Int32 IndexOf(this StringBuilder sb, Char value) {
      for (Int32 index = 0; index < sb.Length; index++)
         if (sb[index] == value) return index;
      return -1;
   }
}
```

Now, when the compiler sees code like the following, the compiler first checks if the StringBuilder class or any of its base classes offers an instance method called IndexOf that takes a single Char parameter.

```
Int32 index = sb.IndexOf('X');
```

If an existing instance method exists, then the compiler produces IL code to call it. If no matching instance method exists, then the compiler will look at any static classes that define static methods called IndexOf that take as their first parameter a type matching the type of the expression being used to invoke the method. This type must also be marked with the this keyword. In this example, the expression is sb, which is of the StringBuilder type. In this case, the compiler is looking specifically for an IndexOf method that takes two parameters: a StringBuilder (marked with the this keyword) and a Char. The compiler will find our IndexOf method and produce IL code that calls our static method.

OK—so this now explains how the compiler improves the last two problems related to code understandability that I mentioned earlier. However, I haven't yet addressed the first problem: how does a programmer know that an `IndexOf` method even exists that can operate on a `StringBuilder` object? The answer to this question is found in Microsoft Visual Studio's IntelliSense feature. In the editor, when you type a period, Visual Studio's IntelliSense window opens to show you the list of instance methods that are available. Well, that IntelliSense window also shows you any extension methods that exist for the type of expression you have to the left of the period. Figure 8-1 shows Visual Studio's IntelliSense window; the icon for an extension method has a down arrow next to it, and the tooltip next to the method indicates that the method is really an extension method. This is truly awesome because it is now easy to define your own methods to operate on various types of objects and have other programmers discover your methods naturally when using objects of these types.

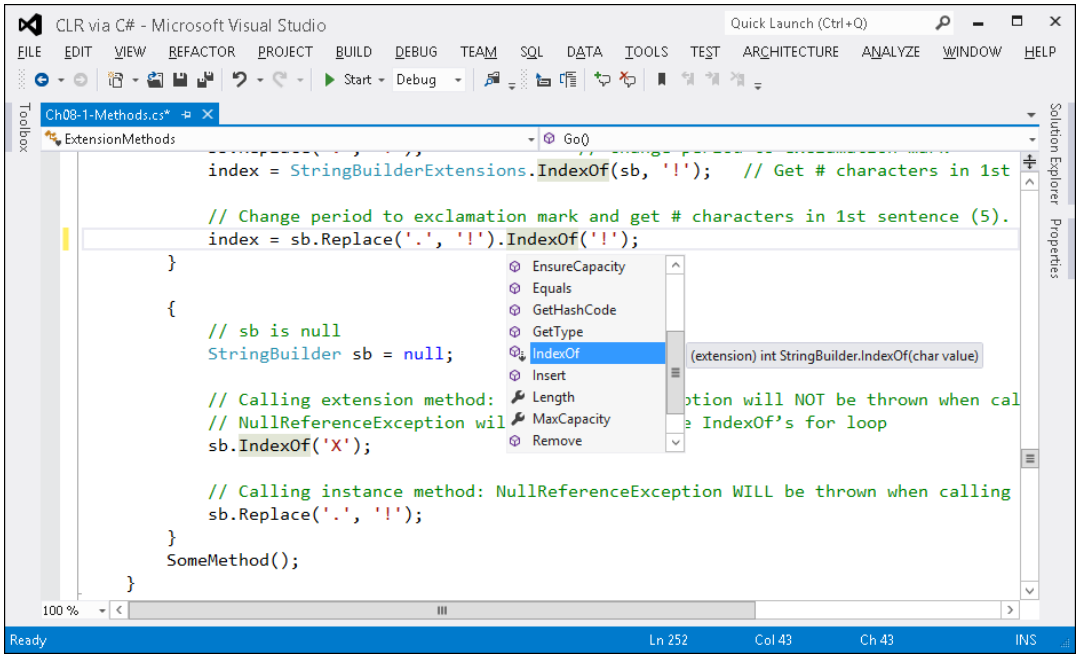

FIGURE 8-1 Visual Studio's IntelliSense window, showing extension methods.

Rules and Guidelines

There are some additional rules and guidelines that you should know about extension methods:

- C# supports extension methods only; it does not offer extension properties, extension events, extension operators, and so on.
- Extension methods (methods with `this` before their first argument) must be declared in nongeneric, static classes. However, there is no restriction on the name of the class; you can call it whatever you want. Of course, an extension method must have at least one parameter, and only the first parameter can be marked with the `this` keyword.

- The C# compiler looks only for extension methods defined in static classes that are themselves defined at the file scope. In other words, if you define the static class nested within another class, the C# compiler will emit the following message: `error CS1109: Extension method must be defined in a top-level static class; StringBuilderExtensions is a nested class.`

- Because the static classes can have any name you want, it takes the C# compiler time to find extension methods because it must look at all the file-scope static classes and scan their static methods for a match. To improve performance and also to avoid considering an extension method that you may not want, the C# compiler requires that you "import" extension methods. For example, if someone has defined a `StringBuilderExtensions` class in a `Wintellect` namespace, then a programmer who wants to have access to this class's extension methods must put a `using Wintellect;` directive at the top of his or her source code file.

- It is possible that multiple static classes could define the same extension method. If the compiler detects that two or more extension methods exist, then the compiler issues the following message: `error CS0121: The call is ambiguous between the following methods or properties: 'StringBuilderExtensions.IndexOf(string, char)' and 'AnotherStringBuilderExtensions.IndexOf(string, char)'.` To fix this error, you must modify your source code. Specifically, you cannot use the instance method syntax to call this static method anymore; instead you must now use the static method syntax where you explicitly indicate the name of the static class to explicitly tell the compiler which method you want to invoke.

- You should use this feature sparingly, because not all programmers are familiar with it. For example, when you extend a type with an extension method, you are actually extending derived types with this method as well. Therefore, you should not define an extension method whose first parameter is `System.Object`, because this method will be callable for all expression types and this will really pollute Visual Studio's IntelliSense window.

- There is a potential versioning problem that exists with extension methods. If, in the future, Microsoft adds an `IndexOf` instance method to their `StringBuilder` class with the same prototype as my code is attempting to call, then when I recompile my code, the compiler will bind to Microsoft's `IndexOf` instance method instead of my static `IndexOf` method. Because of this, my program will experience different behavior. This versioning problem is another reason why this feature should be used sparingly.

Extending Various Types with Extension Methods

In this chapter, I demonstrated how to define an extension method for a class, `StringBuilder`. I'd like to point out that because an extension method is really the invocation of a static method, the CLR does not emit code ensuring that the value of the expression used to invoke the method is not `null`.

```
// sb is null
StringBuilder sb = null;
```

```
// Calling extension method: NullReferenceException will NOT be thrown when calling IndexOf
// NullReferenceException will be thrown inside IndexOf's for loop
sb.IndexOf('X');

// Calling instance method: NullReferenceException WILL be thrown when calling Replace
sb.Replace('.', '!');
```

I'd also like to point out that you can define extension methods for interface types, as the following code shows.

```
public static void ShowItems<T>(this IEnumerable<T> collection) {
   foreach (var item in collection)
      Console.WriteLine(item);
}
```

The extension method above can now be invoked using any expression that results in a type that implements the `IEnumerable<T>` interface.

```
public static void Main() {
   // Shows each Char on a separate line in the console
   "Grant".ShowItems();

   // Shows each String on a separate line in the console
   new[] { "Jeff", "Kristin" }.ShowItems();

   // Shows each Int32 value on a separate line in the console
   new List<Int32>() { 1, 2, 3 }.ShowItems();
}
```

Important Extension methods are the cornerstone of Microsoft's Language Integrated Query (LINQ) technology. For a great example of a class that offers many extension methods, see the static `System.Linq.Enumerable` class and all its static extension methods in the Microsoft .NET Framework SDK documentation. Every extension method in this class extends either the `IEnumerable` or `IEnumerable<T>` interface.

You can define extension methods for delegate types, too. Here is an example.

```
public static void InvokeAndCatch<TException>(this Action<Object> d, Object o)
   where TException : Exception {
   try { d(o); }
   catch (TException) { }
}
```

And here is an example of how to invoke it.

```
Action<Object> action = o => Console.WriteLine(o.GetType()); // Throws NullReferenceException
action.InvokeAndCatch<NullReferenceException>(null);         // Swallows NullReferenceException
```

You can also add extension methods to enumerated types. I show an example of this in the "Adding Methods to Enumerated Types" section in Chapter 15, "Enumerated Types and Bit Flags."

And last but not least, I want to point out that the C# compiler allows you to create a delegate (see Chapter 17, "Delegates," for more information) that refers to an extension method over an object.

```
public static void Main () {
   // Create an Action delegate that refers to the static ShowItems extension method
   // and has the first argument initialized to reference the "Jeff" string.
   Action a = "Jeff".ShowItems;
   .
   .
   .
   // Invoke the delegate that calls ShowItems passing it a reference to the "Jeff" string.
   a();
}
```

In the preceding code, the C# compiler generates IL code to construct an `Action` delegate. When creating a delegate, the constructor is passed the method that should be called and is also passed a reference to an object that should be passed to the method's hidden `this` parameter. Normally, when you create a delegate that refers to a static method, the object reference is `null` because static methods don't have a `this` parameter. However, in this example, the C# compiler generated some special code that creates a delegate that refers to a static method (`ShowItems`) and the target object of the static method is the reference to the "Jeff" string. Later, when the delegate is invoked, the CLR will call the static method and will pass to it the reference to the "Jeff" string. This is a little hacky, but it works great and it feels natural so long as you don't think about what is happening internally.

The Extension Attribute

It would be best if this concept of extension methods was not C#-specific. Specifically, we want programmers to define a set of extension methods in some programming language and for people in other programming languages to take advantage of them. For this to work, the compiler of choice must support searching static types and methods for potentially matching extension methods. And compilers need to do this quickly so that compilation time is kept to a minimum.

In C#, when you mark a static method's first parameter with the `this` keyword, the compiler internally applies a custom attribute to the method and this attribute is persisted in the resulting file's metadata. The attribute is defined in the System.Core.dll assembly, and it looks like this.

```
// Defined in the System.Runtime.CompilerServices namespace
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class | AttributeTargets.Assembly)]
public sealed class ExtensionAttribute : Attribute {
}
```

In addition, this attribute is applied to the metadata for any static class that contains at least one extension method. And this attribute is also applied to the metadata for any assembly that contains at least one static class that contains an extension method. So now, when compiling code that invokes an instance method that doesn't exist, the compiler can quickly scan all the referenced assemblies to know which ones contain extension methods. Then it can scan only these assemblies for static classes that contain extension methods, and it can scan just the extension methods for potential matches to compile the code as quickly as possible.

Note The `ExtensionAttribute` class is defined in the System.Core.dll assembly. This means that the resulting assembly produced by the compiler will have a reference to System.Core.dll embedded in it even if I do not use any types from System.Core.dll and do not even reference System.Core.dll when compiling my code. However, this is not too bad a problem because the `ExtensionAttribute` is used only at compile time; at run time, System.Core.dll will not have to be loaded unless the application consumes something else in this assembly.

Partial Methods

Imagine that you use a tool that produces a C# source code file containing a type definition. The tool knows that there are potential places within the code it produces where you might want to customize the type's behavior. Normally, customization would be done by having the tool-produced code invoke virtual methods. The tool-produced code would also have to contain definitions for these virtual methods, and the way these methods would be implemented is to do nothing and simply return. Now, if you want to customize the behavior of the class, you'd define your own class, derive it from the base class, and then override any virtual methods implementing it so that it has the behavior you desire. Here is an example.

```
// Tool-produced code in some source code file:
internal class Base {
   private String m_name;

   // Called before changing the m_name field
   protected virtual void OnNameChanging(String value) {
   }

   public String Name {
      get { return m_name; }
      set {
         OnNameChanging(value.ToUpper());  // Inform class of potential change
         m_name = value;                   // Change the field
      }
   }
}


// Developer-produced code in some other source code file:
internal class Derived : Base {
   protected override void OnNameChanging(string value) {
      if (String.IsNullOrEmpty(value))
         throw new ArgumentNullException("value");
   }
}
```

Unfortunately, there are two problems with the preceding code:

- The type must be a class that is not sealed. You cannot use this technique for sealed classes or for value types (because value types are implicitly sealed). In addition, you cannot use this technique for static methods because they cannot be overridden.
- There are efficiency problems here. A type is being defined just to override a method; this wastes a small amount of system resources. And, even if you do not want to override the behavior of `OnNameChanging`, the base class code still invokes a virtual method that simply does nothing but return. Also, `ToUpper` is called whether `OnNameChanging` accesses the argument passed to it or not.

C#'s partial methods feature allows you the option of overriding the behavior or a type while fixing the aforementioned problems. The code below uses partial methods to accomplish the same semantic as the previous code.

```
// Tool-produced code in some source code file:
internal sealed partial class Base {
   private String m_name;

   // This defining-partial-method-declaration is called before changing the m_name field
   partial void OnNameChanging(String value);

   public String Name {
      get { return m_name; }
      set {
         OnNameChanging(value.ToUpper());  // Inform class of potential change
         m_name = value;                   // Change the field
      }
   }
}

// Developer-produced code in some other source code file:
internal sealed partial class Base {

   // This implementing-partial-method-declaration is called before m_name is changed
   partial void OnNameChanging(String value) {
      if (String.IsNullOrEmpty(value))
         throw new ArgumentNullException("value");
   }
}
```

There are several things to notice about this new version of the code:

- The class is now sealed (although it doesn't have to be). In fact, the class could be a static class or even a value type.
- The tool-produced code and the developer-produced code are really two partial definitions that ultimately make up one type definition. For more information about partial types, see the "Partial Classes, Structures, and Interfaces" section in Chapter 6, "Type and Member Basics."

- The tool-produced code defined a partial method declaration. This method is marked with the `partial` token and it has no body.
- The developer-produced code implemented the partial method declaration. This method is also marked with the `partial` token and it has a body.

Now, when you compile this code, you see the same effect as the original code I showed you. Again, the big benefit here is that you can rerun the tool and produce new code in a new source code file, but your code remains in a separate file and is unaffected. And, this technique works for sealed classes, static classes, and value types.

Note In Visual Studio's editor, if you type in `partial` and press the spacebar, the IntelliSense window shows you all the enclosing type's defined partial method declarations that do not yet have matching implementing partial method declarations. You can then easily select a partial method from the IntelliSense window and Visual Studio will produce the method prototype for you automatically. This is a very nice feature that enhances productivity.

But, there is another big improvement we get with partial methods. Let's say that you do not need to modify the behavior of the tool-produced type. In this case, you do not supply your source code file at all. If you just compile the tool-produced code by itself, the compiler produces IL code and metadata as if the tool-produced code looked like this.

```
// Logical equivalent of tool-produced code if there is no
// implementing partial method declaration:
internal sealed class Base {
   private String m_name;

   public String Name {
      get { return m_name; }
      set {
         m_name = value;                    // Change the field
      }
   }
}
```

That is, if there is no implementing partial method declaration, the compiler will not emit any metadata representing the partial method. In addition, the compiler will not emit any IL instructions to call the partial method. And the compiler will not emit code that evaluates any arguments that would have been passed to the partial method. In this example, the compiler will not emit code to call the `ToUpper` method. The result is that there is less metadata/IL, and the run-time performance is awesome!

Note Partial methods work similarly to the `System.Diagnostics.ConditionalAttribute` attribute. However, partial methods work within a single type only while the `Conditional-Attribute` can be used to optionally invoke methods defined in another type.

Rules and Guidelines

There are some additional rules and guidelines that you should know about partial methods:

- They can only be declared within a partial class or `struct`.
- Partial methods must always have a return type of `void`, and they cannot have any parameters marked with the `out` modifier. These restrictions are in place because at run time, the method may not exist and so you can't initialize a variable to what the method might return because the method might not exist. Similarly, you can't have an `out` parameter because the method would have to initialize it and the method might not exist. A partial method may have `ref` parameters, may be generic, may be instance or static, and may be marked as `unsafe`.
- Of course, the defining partial method declaration and the implementing partial method declaration must have identical signatures. If both have custom attributes applied to them, then the compiler combines both methods' attributes together. Any attributes applied to a parameter are also combined.
- If there is no implementing partial method declaration, then you cannot have any code that attempts to create a delegate that refers to the partial method. Again, the reason is that the method doesn't exist at run time. The compiler produces this message: `error CS0762: Cannot create delegate from method 'Base.OnNameChanging(string)' because it is a partial method without an implementing declaration`.
- Partial methods are always considered to be private methods. However, the C# compiler forbids you from putting the `private` keyword before the partial method declaration.