CHAPTER 3

.NET Framework and Language Enhancements in 2005

IN THIS CHAPTER

- Shared .NET Language Additions
- VB Language Enhancements
- C# Language Enhancements
- .NET Framework 2.0 Enhancements

The majority of this book focuses on unlocking the productivity promises of the Visual Studio IDE. However, we thought it important to also cover some of the recent advances in the .NET languages and the Framework. These items (the IDE, the languages, and the Framework) all ship from Microsoft in concert. Therefore, any discussion of the new IDE would be incomplete without some mention of the elements that have been bound to it.

This chapter covers the enhancements relative to both Visual Basic .NET and C#. In addition, it highlights some of the key advances made in the Framework. Our assumption is that a majority of readers have some base-level understanding of either VB or a C-based language prior to the current version, along with a decent grasp of the .NET Framework. Therefore, our approach should give you insight into those enhancements that make .NET 2.0 a big leap forward over prior versions.

Shared .NET Language Additions

The .NET languages pick up a number of enhancements as a result of updates made to the common language runtime (CLR). Although there are specific enhancements for both Visual Basic and C#, respectively, the big advancements made in 2005 apply to both languages. Therefore, we will cover them as a group and provide examples in both languages. This group of .NET language enhancements includes the following key additions:

- Generics
- Nullable types
- Partial types
- Properties with mixed access levels
- Ambiguous namespaces

We will cover each of these items in detail in the coming sections. Again, we provide examples in both C# and VB because these enhancements apply to both languages. We will cover the VB and C# language-specific enhancements later in the chapter.

Generics

Generics are undoubtedly the biggest addition to .NET in version 2.0. As such, no book would be complete without covering their ins and outs. Generics may seem daunting at first—especially if you start looking through code that contains strange angle brackets in the case of C# or the `Of` keyword for Visual Basic. The following sections define generics, explain their importance, and show you how to use them in your code.

Generics Defined

The concept of *generics* is relatively straightforward. You need to develop an object (or define a parameter to a method), but you do not know the object's type when you write the code. Rather, you want to write the code *generically* and allow the caller to your code to determine the actual type of the object.

You could simply use the `System.Object` class to accomplish this. That is what we did prior to 2.0. However, imagine you also want to eliminate the need for boxing, runtime type checking, and explicit casting everywhere in your code. Now you can start to see the vision for generics.

The benefits of generics can best be seen through an example. The easiest example is that of creating a collection class that contains other objects. For our example, image you want to store a series of objects. You might do so by adding each object to an ArrayList. However, the compiler and runtime know only that you have some list of objects. The list could contain `Order` objects or `Customer` objects or both (or anything). The only way to know what is contained in the list is to write code to check for the type of the object in the list.

Of course, to get around this issue, you might write your own strongly typed lists. Although this approach is viable, it results in tedious code written over and over for each type you want to work with as a collection. The only real difference in the code is the type allowed in the list. In addition, you still have to do all the casting because the underlying list still simply contains types as `System.Object`.

Now imagine if you could write a single class that, when used, allows the user to define its type. You can then write one *generic* list class that, instead of containing types as `System.Object`, would contain objects as the type with which the class is defined. This

allows a caller to the generic list to decide the list should be of type `Orders` or only contain `Customers`. This is precisely what generics afford us. Think of a generic class as a template for a class.

Generics come in two flavors: generic types and generic methods. *Generic types* are classes whose type is defined by the code that creates the class. A *generic method* is one that defines one or more generic type parameters. In this case, the generic parameter is used throughout the method but its type is defined only when the method is called. In addition, you can define constraints that control the creation of generics. In the coming sections, we'll look at all of these items.

The Benefits of Generics

Now you should plainly see some of the benefits that generics provide. Without them, any class that is written to manage different types must use `System.Object`. This presents a number of problems. First, there is no constraint or compiler checking on what goes into the object. The corollary is also true: You cannot know what you are getting out if you cannot constrain what goes in. Second, when you use the object, you must do type checking to verify its type and then do casting to cast it back to its original type. This, of course, comes with a performance penalty. Finally, if you use value types and store them in `System.Object`, then they get boxed. When you later retrieve this value type, it must be unboxed. Again, this adds unwanted code and unnecessary performance hits. Generics solve each of these issues. Let's look at how this is possible.

How .NET Manages Generics

When you compile a generic type, you generate Microsoft Intermediate Language (MSIL) code and metadata (just like all the rest of your .NET code). Of course, for the generic type or method, the compiler emits MSIL that defines your use of generic types.

With all MSIL code, when it is first accessed, the just-in-time (JIT) compiler compiles the MSIL into native code. When the JIT compiler encounters a generic, it knows the actual type that is being used in place of the generic. Therefore, it can substitute the real type for the generic type. This process is called *generic type instantiation*.

The newly compiled, native type is now used by subsequent, similar requests. In fact, all reference types are able to share a single generic type instantiation because, natively, references are simply pointers with the same representation. Of course, if a new value type is used in the generic type instantiation, the runtime will jit a new copy of the generic type.

This is how we get the benefits of generics both when we're writing our code and when it executes. Upon execution, all our code becomes native, strongly typed code. Now let's look at coding some generics.

Creating Generic Types

Generic types are classes that contain one or more elements whose type should be determined at instantiation (rather than during development). To define a generic type, you first declare a class and then define type parameters for the class. A *type parameter* is one that is passed to a class that defines the actual type for the generic. You can think of a

type parameter as similar to method parameters. The big difference is that, instead of passing a value or a reference to an object, you are passing the type used by the generic.

NOTE

Most generic types are written to manage collections of objects or linked lists. Generics are not, however, limited to just managing collections. Any class you write can use generics.

As an example, suppose you are writing a class called `Fields` that works with name/value pairs similar to a `Hashtable` or `Dictionary`. You might declare the class as follows:

C#

```
public class Fields
```

VB

```
Public Class Fields
```

Let's also suppose that the class can work with a variety of types for its keys and a variety of types for its values. You want to write the class generically to support multiple types. However, after the class is instantiated, you want it to be constrained to the types used to create the class. To add the type parameters to the class declaration, you would then write the following:

C#

```
public class Fields<keyType, valueType>
```

VB

```
Public Class Fields(Of keyType, valueType)
```

In this case, `keyType` and `valueType` are type parameters that can be used in the rest of the class to reference the types that will be passed to the class. For example, you might then have an `Add` method in your class whose signature looks like the following:

C#

```
public void Add(keyType key, valueType value)
```

VB

```
Public Sub Add(key as keyType, value as valueType)
```

This indicates to the compiler that whatever types are used to create the class should also be used in this method. In fact, to consume the class, your code would first create an instance and pass type arguments to the instance. *Type arguments* are the types passed to type parameters. The following is an example:

C#

```
Fields<int, Field> myFields = new Fields<int, Field>();
```

VB

```
Dim myFields As New Fields(Of Integer, Field)
```


In this case a new instance of the generic `Fields` class is created that must contain `int` (`integer`) value for its keys and `Field` instances for its values. Calling the `Add` method of the newly created `Fields` object would then look like this:

C#

```
myFields.Add(1, new Field());
```

VB

```
myFields.Add(1, New Field())
```

If you try to pass another type to either parameter, you will get a compiler error because the object becomes strongly typed at this point.

TIP

When you see generics used, especially in C#, you will often see single letters used for defining type names. It is not uncommon to see <T> or <K>. You are not constrained to these short names. It is always better to provide somewhat more descriptive names.

Creating Generic Methods

So far we've looked at generic type parameters. These type parameters end up defining variables with class-level scope. That is, the variable that defines the generic type is available throughout the entire class. As with any class you write, you may not need class-level scoping. Instead, it may be sufficient to define the elements passed to a given method. Generics are no different in this regard. You can define them at the class level (as we've shown) or at the method level (as we will see).

Generic methods work well for common, utility-like functions that execute a common operation on a variety of similar types. You define a generic method by indicating the existence of one or more generic types following the method name. You can then refer to

these generic types inside the method's parameter list, its return type, and of course, the method body. The following shows the syntax for defining a generic method:

C#

```
public void Save<instanceType>(instanceType type)
```

VB

```
Public Sub Save(Of instanceType)(ByVal type As instanceType)
```

To call this generic method, you must define the type passed to the method as part of the call to the method. Suppose the `Save` method defined in the preceding example is contained in a class called `Field`. Now suppose you have created an instance of `Field` and have stored a reference to it in the variable named `myField`. The following code shows how you might call the `Save` method passing the type argument to the method:

C#

```
myField.Save<CustomerOrder>(new CustomerOrder());
```

VB

```
myField.Save(Of CustomerOrder)(New CustomerOrder())
```

We need to add a few notes on generic methods. First, you can often omit the type parameter when calling a generic method. The compiler can figure out the type based on the parameter passed to it. Therefore, the type parameter is optional when calling a generic method. However, it is generally preferable to pass the type because it makes your code more readable and saves the compiler from having to look it up. Second, generic methods can be declared as static (or shared). Finally, you can define constraints on generic methods (and classes), as we will see in the next section.

Getting Specific with Generics (Constraints)

When you first encounter generic methods, it can be easy to think of them as simple data storage devices. At first glance, they seem to have a huge flaw. This flaw can best be described with the question that might be gnawing at you, "Generics are great, but what if you want to call a method or property of a generic object whose type, by definition, you are unaware of?" This flaw seems to limit the use of generics. However, upon a closer look, you'll see that generic constraints allow you to overcome this perceived flaw.

Generic constraints are just what they sound like: They allow you to define restrictions on the types that a caller can use when creating an instance of your generic class or calling one of your generic methods. Generic constraints have the following three variations:

- **Derivation constraint**—Allows you to indicate that the generic type must implement one or more specific interfaces or derive from a base class.
- **Default constructor constraint**—Allows you to indicate that the generic type must expose a constructor without parameters.
- **Reference/value constraint**—Allows you to indicate that a generic type parameter must either be a reference or a value type.

Using a derivation constraint enables you to indicate one or more interfaces (or object types) that are allowed to be passed to the generic class. Doing so allows you to overcome the aforementioned flaw. For example, if in the `Fields` generic class defined previously you need to be able to call a method or property of the generic `valueType` (perhaps a property that aids in sorting the group of `Fields`), you can now do so, provided that method or property is defined on the interface or base class constraint. The following provides an example of defining a derivation constraint on a generic class:

C# Class Constraint

```
public class Fields<keyType, valueType> where keyType : ISort
```

VB Class Constraint

```
Public Class Fields(Of keyType, valueType As ISort)
```

In the preceding example, the class named `Fields`, which defines the two generic types `valueType` and `keyType`, contains a constraint on `keyType`. The constraint is that `keyType` must implement an interface called `ISort`. This now allows the generic class `Fields` to use methods of `ISort` without casting.

NOTE

You can define a derivation constraint for both generic classes and generic methods.

You can indicate any number of interfaces that the generic type must implement. However, you can indicate only a single base class from which the generic type can derive. You can, of course, pass to the generic type an object that itself inherits from this constraining base class.

NOTE

If you override a generic method in a base class, you *cannot* add (or remove) constraints to the generic method. Only the constraints defined in the base class will apply to the overridden method.

Generic Collections Namespace

Now that you've seen how to create your own generic classes, it is important to note that the .NET Framework provides a number of generic classes for you to use in your applications. The namespace `System.Collections.Generics` defines a number of generic collection classes designed to allow you to work with groups of objects in a strongly typed manner. A *generic collection* is a collection class that allows a developer to specify the type that is contained in the collection when declaring the collection.

> **NOTE**
>
> By default, Visual Studio adds a reference to the namespace `System.Collections.Generics` to all VB and C# code files.

The generic classes defined in this namespace are varied based on their usage. The classes include one called `List` designed for working with a simple list or array of objects. It also includes a `SortedList`, a `LinkedList`, a `Queue`, a `Stack`, and several `Dictionary` classes. These classes cover all the basics of working without strongly typed collection classes. In addition, the namespace also defines a number of interfaces that you can use when building your own generic collections.

Nullable Types

Most of us have written applications in which we were forced to declare a variable and choose a default value prior to knowing what value that variable should contain. For instance, imagine you have a class called `Person` with a Boolean property called `IsFemale`. If you do not implicitly know a person's sex at object instantiation, you are forced to pick a default, or you must implement the property as a tri-state enumeration (or similar) with values `Male`, `Female`, and `Unknown`.

The latter can be cumbersome, especially if the value is stored as a Boolean in the database. There are similar examples. Imagine if you are writing a `Test` class with an integer value called `Score`. If you are unsure of the `Score` value, you end up initializing this variable to zero (0). This value, of course, does not represent a real score. You then must program around this fact by either tracking zero as a magic number or carrying another property like `IsScoreSet`.

These examples are further amplified by the fact that the databases we work with all understand that a value can be null (or not set). We are often unable to use this feature unless we write code to do translation during our insert and select transactions.

Nullable types in .NET 2.0 are meant to free us from these issues. A *nullable type* is a special value type that can have a null assigned to it. This is unlike the value types we are accustomed to (`int`, `bool`, `double`, and so on); these are simply not initialized when declared. On the contrary, with nullable types, you can create integers, Booleans, doubles, and the like and assign them the value of null. You no longer have to guess (or code around) whether a variable has been set. This includes no longer having to provide a

default value. Instead, you now can initialize or assign a variable to the value of null. You can now write code without default assumptions. In addition, nullable types also solve the issue of pushing and pulling nulls to and from the database. Let's look at how they work.

Declaring Nullable Types

Declaring a nullable type is very different between the C# and VB languages. However, both result in declaring the same nullable value type structure inside the .NET Framework (`System.Nullable`). This generic structure is defined by the type that is used in its declaration. For example, if you are defining a nullable integer, the generic structure returns an integer version. The following code snippets demonstrate how nullable types are declared in both C# and VBL:

A C# Nullable Type Example

```
bool? hasChildren = null;
```

A VB Nullable Type Example

```
Dim hasChildren As Nullable(Of Boolean) = Nothing
```

Notice that in the C# example, you can use the `?` type modifier to indicate that a base type should be treated as a nullable type. This is simply a shortcut. It allows developers to use the standard syntax for creating types but simply add a question mark to turn that type to a nullable version. On the contrary, if you are coding in VB, you are required to be more explicit by defining the `Nullable` class as you would a similar generic. You can also use a similar syntax in C#, as in the following example:

```
System.Nullable<bool> hasChildren = null;
```

> **NOTE**
>
> Only value types can be nullable. Therefore, it is not valid to create a nullable string or a developer-defined class. However, you can create nullable instances of structures because they are value types.

Working with Nullable Types

The generic `System.Nullable` structure contains two read-only properties: `HasValue` and `Value`. These properties allow you to work with nullable types efficiently. The `HasValue` property is a Boolean value that indicates whether a given nullable type has a value assigned to it. You can use this property in `If` statements to determine whether a given variable has been assigned. In addition, you can simply check the variable for null (C# only). The following provides an example of each:

C# `HasValue` Example

```
If (hasChildren.HasValue) {…}
```

VB `HasValue` Example

```
If hasChildren.HasValue Then
```

C# Checking the Variable for Null

```
if (hasChildren != null) {…}
```

VB Checking the Variable `Value` for Null

```
If hasChildren.Value <> Nothing Then
```

The `Value` property simply returns the value contained by the `Nullable` structure. You can also access the value of the variable by calling the variable directly (without using the `Value` property). The distinction lies in that when `HasValue` is false, calls to the `Value` property will result in an exception being thrown. Whereas when you access the variable directly in this condition (`HasValue = false`), no exception is thrown. Therefore, it is important to know exactly the behavior you require and use these options correctly. The following provides an example of using the `Value` property:

C# `Value` Property Example

```
System.Nullable<bool> hasChildren = null;
Console.WriteLine(hasChildren);  //no exception is thrown
if (hasChildren != null) {
  Console.WriteLine(hasChildren.Value.ToString());
}
Console.WriteLine(hasChildren.Value); //throws InvalidOperationException
```

VB `Value` Property Example

```
Dim hasChildren As Nullable(Of Boolean) = Nothing
Console.WriteLine(hasChildren)  'no exception is thrown
If hasChildren.HasValue Then
  Console.WriteLine(hasChildren.Value.ToString())
End If
Console.WriteLine(hasChildren.Value)  'throws InvalidOperationException
```

In the preceding example, the call directly to `hasChildren` will not throw an exception. However, when you try to check the `Value` property when the variable is null, the Framework throws the `InvalidOperationException`.

Partial Types (Classes)

Partial types are simply a mechanism for defining a single class, struct, or interface across multiple code files. In fact, when your code is compiled, there is no such thing as a partial type. Rather, partial types exist only during development. The files that define a partial type are merged together into a singe class during compilation.

Partial types are meant to solve two problems. First, they allow developers to split large classes across multiple files. This potentially allows multiple team members to work on the same class without working on the same file (thus avoiding the related code-merge headaches). The other problem partial types solve is to further partition tool-generated code from that of the developer's. This keeps your code file clean (with only your work in it) and allows a tool to generate portions of the class behind the scenes. Visual Studio 2005 developers will immediately notice this when working with Windows forms, Web Service wrappers, ASP code-behind pages, and the like. If you've worked with these items in prior versions of .NET, you'll soon notice that when you're working in 2005, the generated code is now absent and the class that you write has been marked as `partial`.

Working with Partial Types

Partial types are declared as such using the keyword `Partial`. This keyword is actually the same in both C# and VB. You can apply this keyword to classes, structures, and interfaces. If you do so, the keyword must be the first word on the declaration (before `Class`, `Structure`, or `Interface`). Indicating a partial type tells the compiler to merge these items together upon compilation into a single `.dll` or `.exe`.

When defining partial types, you must follow a few simple guidelines. First, all types with the same name in the same namespace must use the `Partial` keyword. You cannot, for instance, declare a class as `Partial Public Person` in one file and then declare that same class as `Public Person` in another file under the same namespace. Of course, to do so, you would add the `Partial` keyword to the second declaration. Second, you must keep in mind that all modifiers of a partial type are merged together upon compilation. This includes class attributes, XML comments, and interface implementations. For example, if you use the attribute `System.SerializableAttribute` on a partial type, the attribute will be applied to all portions of the type when merged and compiled. Finally, it's important to note that all partial types must be compiled into the same assembly (`.dll` or `.exe`). You cannot compile a partial type across assemblies.

Properties with Mixed Access Levels

In prior versions of .NET, you were able to indicate the access level (public, private, protected, internal) only of an entire property. However, often you might need to make the property read (get) public but control the write (set) internally. The only real solution to this problem using prior .NET versions was not to implement the property set. You would then create another internal method for setting the value of the property. It would

make your coding easier to write and understand if you had fine-grained control over access modifiers of your properties.

.NET 2.0 gives you control of the access modifiers at both the `set` and `get` methods of a property. Therefore, you are free to mark your property as public but make the set private or protected. The following code provides an example:

C# Mixed Property Access Levels

```
private string _userId;
public string UserId {
  get { return _userId; }
  internal set { userId = value; }
}
```

VB Mixed Property Access Levels

```
Private _userId As String
Public Property UserId() As String
  Get
    Return _userId
  End Get
  Friend Set(ByVal value As String)
    _userId = value
  End Set
End Property
```

Ambiguous Namespaces

On large projects, it is possible to easily run into namespace conflicts with each other and with the .NET Framework (`System` namespace). Previously, these ambiguous references were not resolvable. Instead, you got an exception at compile time.

.NET 2.0 now allows developers to define a `System` namespace of their own without blocking access to the .NET version. For example, suppose you define a namespace called `System` and suddenly are unable to access the global version of `System`. In C# you would add the keyword `global` along with a namespace alias qualifier `::` as in the following syntax:

```
global::System.Double myDouble;
```

In VB the syntax is similar but uses the keyword `Global`:

```
Dim myDouble As Global.System.Double
```

To further manage namespace conflict, you can still define an alias when using (or importing) a namespace. This alias can then be used to reference types within the

namespace. For example, suppose you had a conflict with the `System.IO` namespace. You could define an alias upon import as follows:

C#

```
using IoAlias = System.IO;
```

VB

```
Imports IoAlias = System.IO
```


You could then reference types by using the alias directly. Of course, Visual Studio still gives you complete IntelliSense on these items. The following provides an example of using the alias defined in the preceding example. Notice the new syntax that is possible in C# with the double colon operator:

C# new syntax

```
IoAlias::FileInfo file;
```

C# old syntax

```
IoAlias.FileInfo file;
```

VB

```
Dim file as IoAlias.FileInfo
```

VB Language Enhancements

Former VB developers will be pleased to find that the edit-and-continue feature is back in Visual Studio 2005! However, that's really an IDE feature. In fact, many new IDE features are considered language-specific. We intend to cover most (if not all) of them throughout the book. The IDE enhancements specific to VB include all of the following:

- Developing with My
- Edit-and-continue
- Code snippets
- IntelliSense enhancements
- Attribute editing in the Properties window
- Error correction and warning
- Exception Assistant

- XML documentation
- Document Outline window
- Project Designer
- Settings Designer
- Resource Designer

These enhancements (and more) help make VB great. However, in the following sections, we intend to focus on the language of VB. We want to point out the VB-specific additions that are so compelling in the 2005 release.

The `Continue` Statement

The new `Continue` statement in VB allows developers to skip to the next iteration in a loop. You use the `Continue` statement in combination with either `Do`, `For`, or `While` depending on the type of loop you're working with. If you want to short-circuit the loop and skip immediately to the next iteration, you simply use `Continue For¦Do¦While`, as in the following example:

```
Sub ProcessCustomers(ByVal customers() As Customer)
  Dim i As Integer
  For i = 0 To customers.GetUpperBound(0)
    If customers(i).HasTransactions = False Then Continue For
    ProcessCustomer(customers(i))
  Next
End Sub
```

Unsigned Types

Visual Basic developers can now use unsigned integer data types (`UShort`, `UInteger`, and `ULong`). In addition, the latest version of VB provides the signed type `SByte`. These new types allow VB developers to more easily call functions in the Windows API because these functions often take and return unsigned types. However, these unsigned types are not supported by the common language specification (CLS). Therefore, if you write code that uses these new types, CLS-compliant code may not be able to work with this code.

`IsNot` Operator

The new `IsNot` operator in VB allows developers to determine whether two objects are the same. Of course, VB developers could do this in prior versions by combining `Not` and `Is` as in `If Not myCustomer Is Nothing`. However, VB developers can now use the less awkward syntax of the `IsNot` operator, as in the following line of code. Note that this example is functionally equivalent to using the prior `Not … Is` syntax.

```
If cust IsNot Nothing Then
```

`Using` Block

VB developers who have spent some time with C# will undoubtedly love the capability to define an object's scope with a `Using` block. With this block, C# developers have been able to guarantee disposal of a resource when the application's execution left a given block for any reason. Good news: This feature has now been added to VB. Suppose, for example, that you want to open a connection to a database. You can now do so with the `Using` block. This way, when execution leaves this block for any reason, the object defined by the `Using` statement (SQL connection object) will be disposed of properly. The following code illustrates this new feature:

```
Using cnn As New System.Data.SqlClient.SqlConnection(cnnStr)
  'place code to use the sql connection here
End Using
```


Form Access Similar to VB6

Developers familiar with Visual Basic version 6 (prior to .NET) will recall having direct access to a form's properties and methods simply by using its name. In prior versions of .NET, developers were forced to create an instance of the form to access its properties. In VB8, developers can once again access a form's members by using its name directly.

Explicit Zero Lower Bound on an Array

In past incarnations of VB (prior to .NET) developers could indicate the upper and lower bounds of an array using the `To` keyword. Developers were able to define an array as starting at 1 and going "to" 10 for instance. This made code that used arrays very easy to read. However, with the advent of .NET and the common language specification (CLS), arrays were forced as zero (0) lower bounds. That is, every array in .NET starts with a zero element. This doesn't change with VB8. However, the ability to define your arrays as starting at 0 and going "to" an upper bound is back, simply for code readability. Therefore, you can define arrays as in the following line of code, but you must define the lower bound as zero (0):

```
Dim myIntArray(0 To 9) As Integer
```

Operator Overloading

If you have written class libraries long enough, you eventually need to define the behavior of your class when used with an operator such as addition (`+`), subtraction (`-`), multiplication (`*`), greater than (`>`), or similar. For example, if you need to calculate the result of how two versions of your class are added together with the `+` operator, you need to define the `+` operator behavior in your code. In prior versions of VB, you could not do this. VB8 allows for what is called *operator overloading*.

You define a new operator by using the keyword `Operator` (in place of `Sub` or `Function`), followed by the operator symbol you intend to overload (`+`, `&`, `*`, `<>`, and so on). You can

then write this "function" as you would any other. It can take parameters and return a value. As an example, if you were going to define how two versions of your object are added together, you would define a + operator that took each version as a parameter and returned a third version as the result. The following code illustrates this structure:

```
Public Operator +(ByVal obj1 As MyObject, ByVal obj2 As MyObject) As MyObject
  'calculate objects and return a new version
End Operator
```

Custom Events

Visual Basic developers are now given control of what happens when delegates are registered with a given developer-defined event. VB has added the keyword `Custom` for use when declaring an event. When you use this keyword to declare an event, you are then required to define accessors for `AddHandler`, `RemoveHandler`, and `RaiseEvent`. These accessors override the default behavior of an event with your own custom code. This capability is useful in situations in which you want all your events to be fired asynchronously, or you need finite control over these operations.

C# Language Enhancements

The C# language takes another step forward in the 2005 release. We have already pointed out some of the common enhancements such as generics and nullable types. In addition, there are new IDE features for the C# developer. Some of these features include the following:

- Code snippets
- Refactoring
- IntelliSense updates
- Code wizards
- Project properties

We will cover those features throughout the book. However, here we intend to focus on C#-specific enhancements for 2005.

TIP

For more information on the C# language and the topics discussed here, Microsoft has created the "C# Language Specification 2.0." This Microsoft Word document is available for download at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs05/html/cs3spec.asp. At this same URL, you can find the complete C# language reference and set of tutorials. In addition, the C# 3.0 specification is already out and ready for review and feedback.

Anonymous Methods

The term *anonymous method* sounds a bit daunting when you first come across it. However, an anonymous method is simply an unnamed block of code (not a method) that is passed directly to a delegate. First, this feature is available only to C# programmers. Second, it is useful only when you do not need the full power of a delegate—that is, when you do not require multiple listeners nor the ability to control (add and remove) who's listening.

The fastest way to understand anonymous methods is to compare the established, standard way of implementing a delegate to using an anonymous method. In prior versions of C#, to use a delegate, you had to write a method that was called by the delegate. This required you to both write a new method and connect that method to the delegate.

For an example, let's look at the way to connect code to a button's (`System.Windows.Forms.Button`) `Click` event. First, the `Click` event is a `System.EventHandler` (or delegate). You want to make sure that code you write in a method is connected to that delegate. Suppose the code is in a method that looks as follows:

```
private void button1_Click(object sender, EventArgs e) {
  label1.Text = "textBox.Text";
}
```

You then connect the method to the delegate. Of course, Visual Studio does the work for you behind the scenes. But you can also write this code manually. In addition, Visual Studio only takes care of connecting UI control delegates (events) to the methods. You are responsible for wiring up other delegates—whether they are custom or part of the framework. The following shows how Visual Studio connects the `button1_Click` method to the `Button` class's `Click` event:

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

As you can see, you have to both write a method for the code and connect that method to the delegate. Now let's look at what is possible with anonymous methods. As we've stated, you can simply pass code directly to the delegate. Therefore, you could add the following line of code to the form's constructor (after the call to `InitializeComponents`):

```
this.button1.Click += delegate {
  label1.Text = "Goodbye";
};
```

As you can see in the example, using an anonymous method involves using the keyword `delegate`. Of course, delegates can take parameters, so there is an optional parameter list after `delegate` (not shown in the example). Finally, there is the statement list (or block of code). This is the code passed anonymously (without a method name) to the delegate. This code is set off by curly braces.

NOTE

Anonymous methods have access to the variables that are in scope from the point where the anonymous method is created. These variables are called *outer variables* of the anonymous method (because the anonymous method itself can define its own *inner variables*). You need to know that by using these outer variables inside an anonymous method, they are considered *captured* by the anonymous method. This means that the lifetime of these captured variables is now dependent on the delegate being *garbage collected* (and not the method).

This section simply introduces anonymous methods. With them, you can write some reasonably sophisticated code (which can also be difficult to understand). As you might imagine, passing lines of code as parameters requires some careful thinking to stay out of trouble.

Static Classes

The new version of the .NET Framework provides language developers support for static classes. A *static class* is one whose every member is declared as a noninstance member (or static). That is, consumers of the class do not have to create an instance of the class to call its members. In fact, the Framework ensures that consumers cannot instantiate a static class. Static classes are common to the .NET Framework; however, the VB language does not currently allow for them. The C# language does. This is where we will focus our static class examples.

NOTE

You can approximate a static class in VB by creating a class with a private constructor. In addition, you would mark all the members on the class as Shared. The drawback is that you do not get compiler enforcement or the capability to use a static constructor.

Defining a Static Class

You create a static class by applying the `Static` keyword to the class declaration. The following line of code provides an example:

```
static class ProjectProperties { …
```

As indicated, declaring a class as static ensures that it cannot be instantiated. You still must explicitly declare all members of the class as static (they are not assumed as such). However, a static class will allow the compiler to verify that no instance members are added to the class by accident. You will receive a compiler error if you place a nonstatic member inside a static class; this includes both public and private members. Listing 3.1 provides a simple example of a static class and its members.

LISTING 3.1 A Static Class

```
namespace StaticClasses {
    static class ProjectProperties {
        static string _projectName;
        static ProjectProperties() {
            _projectName = "SomeNewProject";
        }
        public static string Name {
            get { return _projectName; }
        }
        public static DateTime GetDueDate() {
            //get the date the project is due
            DateTime dueDate = DateTime.Now.AddDays(10);
            return dueDate;
        }
    }
}
```


NOTE

Static classes are automatically sealed. That is, you cannot inherit or derive from a static class.

Constructors and Static Classes

You cannot create a constructor for a static class. However, if you need similar features of a constructor (like setting initial values), you can create what is called a *static constructor*. Listing 3.1 shows an example of the static constructor named `ProjectProperties`. Notice that this constructor initializes the value of the static member `_projectName`.

The .NET CLR loads the static class automatically when the containing namespace is loaded. In addition, when a static member is called, the static constructor is automatically called by the CLR. No instance of the class is necessary (or even possible) to call this special type of constructor.

Reference Two Versions of the Same Assembly

As a developer, you sometimes get stuck between needing the features of an older version of a component and wanting to upgrade to the latest version of that component. Often, this is the result of a third-party component that has evolved without concern for backward compatibility. In these cases, your options are limited to either a complete upgrade to the new component or sticking with the older version. C# 2.0 now provides an additional option: working with both versions through an external assembly alias.

The principal issue with working with multiple versions of the same assembly is resolving conflicts between the names of members that share the same namespace. Suppose, for

instance, that you are working with an assembly that generates charts for your application. Suppose that the namespace is `Charting` and there is a class called `Chart`. When a new version of the assembly is released, you want to be able to keep all of your existing code as is but reference the new assembly for your new code. To do so in C# 2.0, you must follow a couple of steps.

First, you must define an alias for the newly referenced assembly. You do this through the Properties window for the selected reference. Figure 3.1 provides an example of setting this alias. Note that we are setting an alias for version 2 of the assembly (`ChartV2`). This ensures that calls to `Charting.Chart` will still point to version 1 of the assembly (`ChartV1`).

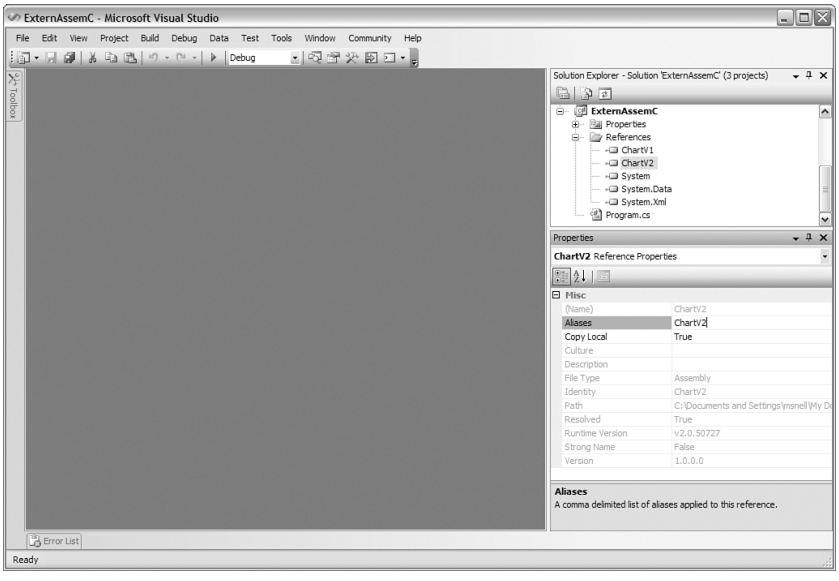

FIGURE 3.1 Defining a reference alias.

Next, in the code file where you plan to use the additional version of the assembly, you must define the external alias. You do so at the top of the file (before the `Using` statements) with the keyword `extern`. The following line shows an example of what would be placed at the top of the file to reference the second version of the `Charting` component:

```
extern alias ChartV2;
```

Finally, to use the members of the new version, you use the `::` operator (as you would for another, similar C# alias). The following line of code provides an example:

```
ChartV2::Charting.Chart.GenerateChart();
```

Friend Assemblies

C# 2.0 allows you to combine assemblies in terms of what constitutes internal access. That is, you can define internal members but have them be accessible by external assemblies. This capability is useful if you intend to split an assembly across physical files but still want those assemblies to be accessible to one another as if they were internal.

> **NOTE**
>
> Friend assembles *do not* allow for access to private members.

You use the new attribute class `InternalsVisibleToAttribute` to mark an assembly as exposing its internal members as friends to another assembly. This attribute is applied at the assembly level. You pass the name and the public key token of the external assembly to the attribute. The compiler will then link these two assemblies as friends. The assembly containing the `InternalsVisibleToAttribute` will expose its internals to the other assembly (and not vice versa). You can also accomplish the same thing by using the command-line compiler switches.

Friend assemblies, like most things, come at a cost. If you define an assembly as a friend of another assembly, the two assemblies become coupled and need to coexist to be useful. That is, they are no longer a single unit of functionality. This can cause confusion and increase management of your assemblies. It is often easier to stay away from this feature unless you have a very specific need.

.NET Framework 2.0 Enhancements

Because there are so many new features in the .NET Framework, we could not begin to cover them in this limited space. Of course, we will do our best to point them out throughout the book. That said, we wanted to make sure to highlight some of those key enhancements that make this version of the .NET Framework such a great advancement. The following outlines a few of these items:

- **64-Bit support**—You can now compile your .NET application to target a 64-bit version of the operating system. This includes native support and WOW64 compatibility support (allows 32-bit applications to run on 64-bit systems).
- **ACL support**—.NET developers can now use access control list (ACL) features to manage permissions on resources from their code. New classes have been added to the IO namespace (and others) to help you grant users rights to files and the like.
- **Authenticated streams**—The new `NegotiateStream` class allows for secure (SSL encrypted) authentication between a client and a server (listener) when transmitting information across the wire. With it, you can securely pass the client's credentials through impersonation or delegation. In addition, the new `SslStream` class allows for the encryption of the data during the transfer.

- **Data Protection API (DPAPI)**—There is new support in .NET 2.0 for working with the DPAPI. Support includes the ability to encrypt passwords and connection strings on the server. Developers could tap into this in prior versions of .NET through a wrapper class available for download. In 2.0, this access is built into the Framework.
- **Network change discovery**—Your application can now be notified when it loses a connection to the network. With the `NetworkChange` class, developers can know when the computer hosting their application has lost its wireless connection or changed its IP address.
- **FTP support**—The `System.Net` namespace now provides classes for working with FTP. Developers can use the `WebRequest`, `WebResponse`, and `WebClient` classes to send and receive files over this protocol.
- **Globalization enhancements**—The new version of the Framework enables developers to define their own custom cultures. This gives you the ultimate flexibility when working with culture-related information in your application. In addition, .NET 2.0 provides updated Unicode support.
- **Greater caching control**—Developers can now use the `System.Net.Cache` namespace to programmatically control caching.
- **Serial I/O device support**—There is now a `SerialPort` class in the `System.IO` namespace. This class allows developers to work with devices that connect to a serial port on a computer.
- **Enhanced SMTP support**—The `System.Net.Mail` namespace enables developers to send email through an SMTP server.
- **Transactions**—.NET developers have a new `System.Transactions` namespace that allows .NET-developed classes to easily participate in a distributed transaction using the Microsoft Distributed Transaction Coordinator (MSDTC).

New Features of Core Technologies

ADO.NET, ASP.NET, and WinForms all have major advancements in the 2005 release. Each of these topics could be the subject of a separate book. ADO.NET, for instance, now has the support for user-defined type (UDT) and asynchronous database operations. Both ASP and WinForms have many new controls (and enhancements to the old controls). Both technologies, for example, bring back zero-code data binding. We suggest you explore each of these items in depth to see the many new advancements in these core areas.

Summary

This chapter presented those core enhancements to the .NET languages that are key to you, as a developer, in writing more and better code during your development day. Advancements such as generics will help you ensure type safety in collections and reduce

error rates as a result. Similarly, nullable types will allow you to code without forcing values into unassigned variables and then coding around these "magic numbers." These and similar advancements made to both the C# and VB languages work to further evolve your toolset and increase your productivity.

Finally, this chapter briefly covered some of the new items inside the .NET Framework. Clearly, there is a lot that is new. The Framework is becoming so large that developers (and books) are often forced to specialize in a particular area. We suggest that you look at our list of enhancements and then jump off to your own specialty area for further exploration.