Chapter 11

Generics

In this chapter:

Unless you are absolutely new to Microsoft .NET programming—or you're a .NET developer who has lived on a desert island for the last two years—you should have heard about generics and the fact that they are the most important addition to Microsoft Visual Basic and other .NET languages. In this chapter, I show that generics are indeed a very important new feature of your favorite language and illustrate several examples of what generics can do to make your code faster, more concise, and more elegant.

In a nutshell, generics give you the ability to define a class that takes a type as an argument. Depending on the type argument, the generic definition generates a different concrete class. In this sense, generics add a degree of polymorphism, much like other techniques based on inheritance, interfaces, or late binding. But you'll soon discover that generics are much, much more powerful.

Before we dive into the topic, bear in mind that generics aren't a completely new concept in the programming world. In fact, .NET generics are similar to C++ templates, so you might already be familiar with the underlying concepts if you've worked in that language before. However, .NET generics have several features and advantages that C++ templates don't, for example, constraints.

Note To avoid long lines, code samples in this chapter assume that the following Imports statements are used at the file or project level:

```
Imports System.Collections
Imports System.Collections.Generic
```

The Need for Generics

Let's start with a classic example that shows why generics can be so useful. Let's consider the ArrayList type, defined in the System.Collections namespace. I cover this and other collection-

like types in Chapter 13, "Arrays and Collections," but for now it will suffice to see how you can define such a collection and add elements to it:

```
' This collection will contain only integer numbers.
Dim col As New ArrayList()
col.Add(11): col.Add(13): col.Add(19)
For Each n As Integer in col
   Console.WriteLine(n)
Next
' Reading an element requires a CType or CInt operator (if Option Strict is On).
Dim element As Integer = CType(col(0), Integer)
```

As simple as it is, this code has a couple of serious problems, one related to robustness and the other related to performance. The former problem is quite simple to demonstrate: the ArrayList was designed to store values of any kind, hence it stores its value internally inside System.Object slots. This means that a developer using the ArrayList can accidentally or purposely add an element that isn't an integer, an action that would make the For Each loop fail at run time:

```
' Adding a string to the collection doesn't raise any compile-time error…
col.Add("abc")
' …but it makes the following statement fail at run time.
For Each n As Integer in col
   Console.WriteLine(n)
Next
```

Also, the latter problem depends on the ArrayList using System.Object variables internally and manifests itself when you use the ArrayList to store value-typed elements, such as numbers, enumerated values, DataTime values, and any user-defined structure. In fact, when you store a value-typed element in an Object variable, the element must be boxed. As you can recall from the section titled "Reference Types and Value Types" in Chapter 2, "Basic Language Concepts," a box operation takes both CPU cycles and memory from the managed heap, and therefore it should be avoided if possible.

The Traditional Solution

Under previous versions of the .NET Framework you can solve the former problem and make the code more robust by defining a new class that inherits from the CollectionBase type, also in the System.Collections namespace. This type is one of the many abstract types provided in the .NET Framework with the purpose of enabling developers to define their own strong-typed collection classes. Here's a very simple implementation of a custom collection class that can store only integers:

```
Public Class IntegerCollection
   Inherits CollectionBase

   Public Sub Add(ByVal item As Integer)
      Me.List.Add(item)
   End Sub
```

```
   Public Sub Remove(ByVal item As Integer)
      Me.List.Remove(item)
   End Sub

   Default Public Property Item(ByVal index As Integer) As Integer
      Get
         Return CType(Me.List(index), Integer)
      End Get
      Set(ByVal Value As Integer)
         Me.List(index) = Value
      End Set
   End Property
End Class
```

The code is quite simple: each method of your IntegerCollection class takes or returns an Integer value and delegates to a method with the same name as the inner IList object named List. In spite of its simplicity, this solution isn't exactly concise: a real-world class that exposes common methods such as Sort, Find, or Reverse (and all their overloads) would take about a hundred lines. Worse, you'd need a distinct class for each different type of strong-typed collection in your application; for example, a DoubleCollection class to hold Double values, a DateTimeCollection class for DateTime values, and so forth. Granted, you can easily generate these collections by taking a template and performing a search-and-replace operation, but for sure you can think of many other, more pleasant ways to spend your time.

All the code you put in the IntegerCollection class makes the application more robust and slightly less verbose because any attempt to store a noninteger value in the collection is trapped at compile time. Also, reading an element doesn't require a CType operator any longer:

```
' This is the only statement that must be changed from the previous example.
Dim col As New IntegerCollection
…
' Reading an element doesn't require any conversion operator.
Dim element As Integer = col(0)
' Adding anything but an integer raises a compile-time error.
col.Add("abc")                  ' *** This statement doesn't compile.
```

However, the IntegerCollection type doesn't resolve the problem related to performance because integer values are still boxed when they are stored in the inner collection. In fact, this approach makes performance slightly worse because each call to a method in the IntegerCollection class must be routed to the method of the inner List collection.

The Generics-Based Solution

The .NET Framework comes with a new namespace named System.Collections.Generic, which contains several generic collections that can be specialized to contain only values of a given type. For example, see how you can define a collection containing only integer values by means of the new List type:

```
 ' This collection will contain only integer numbers.
Dim col As New List(Of Integer)
```

```
col.Add(11): col.Add(13): col.Add(19)
For Each n As Integer in col
   Console.WriteLine(n)
Next
' Reading an element doesn't require any conversion operator.
Dim element As Integer = col(0)
```

The new Of keyword specifies that the generic List type must be specialized to work with elements of Integer type, and only with that type of element. In fact, assuming that Option Strict is On, any attempt to add elements of a different type raises a compile-time error:

```
' Adding a string causes a compile-time error.
col.Add("abc")                  ' *** This statement doesn't compile.
```

Even if this isn't apparent when looking at the code, the solution based on generics also solves the performance problem because the List(Of Integer) collection stores its elements in Integer slots—in general, in the variables typed after the type specified by the Of clause—and therefore no boxing occurs anywhere.

You can easily prove this point by compiling the following sample code:

```
Dim al As New ArrayList
al.Add(9)
Dim list As New List(Of Integer)
list.Add(9)
```

Here's the corresponding IL code generated by the Visual Basic compiler:

```
//000004:       Dim al As New ArrayList
  IL_0001:  newobj     instance void

[mscorlib]System.Collections.ArrayList::.ctor()
  IL_0006:  stloc.0
//000005:       al.Add(9)
  IL_0007:  ldloc.0
  IL_0008:  ldc.i4.s   9
  IL_000a:  box        [mscorlib]System.Int32
  IL_000f:  callvirt   instance int32
            [mscorlib]System.Collections.ArrayList::Add(object)
  IL_0014:  pop

//000006:       Dim list As New List(Of Integer)
  IL_0015:  newobj     instance void class
            [mscorlib]System.Collections.Generic.List`1<int32>::.ctor()
  IL_001a:  stloc.1
//000007:       list.Add(9)
  IL_001b:  ldloc.1
  IL_001c:  ldc.i4.s   9
  IL_001e:  callvirt   instance void class
            [mscorlib]System.Collections.Generic.List`1<int32>::Add(!0)
```

It isn't essential that you understand the meaning of each IL statement here; the key point is that it requires a box IL opcode (in bold type) to prepare the integer value for being passed to

the Add method of the ArrayList object, whereas no such opcode is used when calling the Add method of the List(Of Integer) object.

Because of the missing box operation, adding value-typed items to a generic collection is remarkably faster than is adding the same items to a nongeneric collection, even though the difference can go unnoticed until the repeated box operations cause a garbage collection. In an informal benchmark, adding one million integers to a List object is about six times faster than adding them to an ArrayList is.

You can extract elements from a generic collection and assign them to a strong-typed variable without having to convert them and without causing an unbox operation. This additional optimization can make your read operations faster by a factor of about 30 percent. This speed improvement isn't as impressive as the one that results when you add items, but on the other hand, it occurs even with small collections that don't stress the garbage collector.

The .NET Framework exposes many generic types in addition to the List object just shown: the Dictionary(Of K,V) and SortedDictionary(Of K,V) generic collections enable you to create strong-typed hash tables; the Stack(Of T), Queue(Of T), and LinkedList(Of T) are useful for creating more robust and efficient versions of other common data structures. I cover these and other generic types later in this chapter and in Chapter 13.

Another important note: the type argument you pass when defining a generic instance can be any .NET type, including another generic or nongeneric collection. You can even pass a type that represents an array:

```
' A collection of generic dictionaries
Dim list As New List(Of Dictionary(Of String, Integer))
' A collection of arrays of Integers
Dim arrays As New List(Of Integer())
' Add an array to the collection.
Dim arr() As Integer = {1, 3, 5, 7, 9}
arrays.Add(arr)
' Display the second element of the first array, and then modify it.
Console.WriteLine(arrays(0)(1))             ' => 3
arrays(0)(1) = 999
```

Authoring Generic Types

In addition to using generic types defined in the .NET Framework, Microsoft Visual Basic 2005 also enables you to create your own generic types. As you'll see in a moment, the syntax for doing so is quite intuitive, even though you must account for some nonobvious details.

Generic Parameters

Let's begin with a very simple task: create a strong-typed collection that doesn't allow you to remove or modify an element after you've added it to the collection. The .NET Framework exposes many collection-like types, but none of them has exactly these features. The simplest

thing to do is author a generic type named ReadOnlyList and reuse it to store elements of any sort. For simplicity's sake, the ReadOnlyList type uses a private array whose max number of elements must be defined when you instantiate a new collection:

```
Public Class ReadOnlyList(Of T)
   Dim values() As T
   ' The constructor takes the maximum number of elements.
   Public Sub New(ByVal elementCount As Integer)
      ReDim values(elementCount - 1)
   End Sub

   ' The Count read-only property
   Private m_Count As Integer

   Public ReadOnly Property Count() As Integer
      Get
         Return m_Count
      End Get
   End Property

   ' Add a new element to the collection; error if too many elements.
   Public Sub Add(ByVal value As T)
      values(m_Count) = value
      m_Count += 1
   End Sub

   ' Return the Nth element; error if index is out of range.
   Default Public ReadOnly Property Item(ByVal index As Integer) As T
      Get
         If index < 0 OrElse index >= m_Count Then _
            Throw New ArgumentException("Index out of range")
         Return values(index)
      End Get
   End Property
End Class
```

The key point in the preceding code is the declaration of the generic parameter in the first line by means of the Of keyword:

```
Public Class ReadOnlyList(Of T)
```

Once you have defined the generic parameter, you can reuse it anywhere in the class (as well as in any nested class) as if it were a regular type name. For example, the generic parameter T appears in the declaration of the inner values array and in the signature of the Add and Item members (in bold type). You can use the ReadOnlyList generic type as you'd use the List generic type, except that you must provide the maximum number of elements and you can't remove or modify any element after you've added it:

```
' This read-only list can contain up to 1,000 integer values.
Dim roList As New ReadOnlyList(Of Integer)(1000)
roList.Add(123)
Console.WriteLine(roList(0))                    ' => 123
' *** Next statement causes a compilation error: "Property Item is readonly."
roList(0) = 234
```

When you work with generics, you need a way to distinguish a generic type such as List(Of T), which contains one or more type parameters, from a generic type such as List(Of Integer), where the type parameter has been replaced (or *bound*) to a specific type. A type of the former kind is known as *generic type definition*, *open generic type*, or *unbound generic type*, whereas a type of the latter type is known as *bound generic type*.

Interestingly, the T generic parameter can be reused to define or instantiate other generic types. For example, you can simplify the ReadOnlyList class by using a private List(Of T) object instead of an array; incidentally, this change relieves you of the requirement of passing the maximum number of elements to the constructor:

```
Public Class ReadOnlyList2(Of T)
   Dim values As List(Of T)

   ' The constructor can take the maximum number of elements. (Default value is 16.)
   Public Sub New(Optional ByVal elementCount As Integer = 16)
      values = New List(Of T)(elementCount)
   End Sub

   ' The Count read-only property
   Public ReadOnly Property Count() As Integer
      Get
         Return values.Count
      End Get
   End Property

   ' Add a new element to the collection.
   Public Sub Add(ByVal value As T)
      values.Add(value)
   End Sub

   ' Return the Nth element; error if index is out of range.
   Default Public ReadOnly Property Item(ByVal index As Integer) As T
      Get
         Return values(index)
      End Get
   End Property
End Class
```

The first problem you face when working with generics is that you can't really make any assumption on the type that will be passed to the generic type parameter. For example, the Add method receives an element of the generic type T, but it can't invoke any method on this element except those inherited from System.Object. For the same reason, you can't use any operator on an element of type T, including math and comparison operators, the Is operator, and the IsNot operator. For example, the simplest way to test a value against Nothing is by means of the Object.Equals static method:

```
   Public Sub Add(ByVal value As T)
      ' Add only nonnull elements to the collection.
      If Not Object.Equals(value, Nothing) Then values.Add(value)
   End Sub
```

(Notice that value types can't be equal to Nothing; therefore, the Then statement is always executed if you pass a value type.) Because of these limitations and the inability to invoke methods in the type referenced by the parameter T, generics are best used as *containers* for objects that don't have an active role. Later in this chapter, you'll learn how you can use constraints to be able to invoke members on contained objects.

Multiple Generic Parameters

A generic class can also take multiple generic parameters. For example, consider the following Relation type, a simple class that enables you to create a one-to-one relation between two instances of a given type:

```
Public Class Relation(Of T1, T2)
   Public ReadOnly Object1 As T1
   Public ReadOnly Object2 As T2

   Public Sub New(ByVal obj1 As T1, ByVal obj2 As T2)
      Me.Object1 = obj1
      Me.Object2 = obj2
   End Sub
End Class
```

In spite of its simplicity, the Relation class can be quite useful to expand your object hierarchy with new features. For example, let's say that you have defined a Person class (which holds personal data about an individual) and a Company type (which holds information about a company). The Relation type enables you to indicate for which company a given person works:

```
Dim ca As New Company("Code Architects")
Dim john As New Person("John", "Evans")
Dim relJohnCa As New Relation(Of Person, Company)(john, ca)
Dim ann As New Person("Ann", "Beebe")
Dim relAnnCa As New Relation(Of Person, Company)(ann, ca)
```

In a real program, you typically deal with many persons and many companies, so you'd be better off creating a strong-typed list that can contain Relation objects. This can be achieved by using nested Of keywords:

```
Dim relations As New List(Of Relation(Of Person, Company))
relations.Add(relJohnCa)
relations.Add(relAnnCa)
```

The ability to nest Of keywords is a very powerful technique that extends the power of generics remarkably. For example, the following code extracts all the persons who work for a given company:

```
Function GetEmployees(ByVal relations As List(Of Relation(Of Person, Company)), _
      ByVal company As Company) As List(Of Person)
   Dim result As New List(Of Person)
   For Each rel As Relation(Of Person, Company) In relations
```

```
      If rel.Object2 Is company Then result.Add(rel.Object1)
   Next
   Return result
End Function
```

You might continue the previous example by extracting all the employees of the Code Architects company, as follows:

```
For Each p As Person In GetEmployees(relations, ca)
   Console.WriteLine(p.FirstName & " " & p.LastName)
Next
```

As you can see, using nested Of keywords can make your code quite contorted and nearly unreadable. The following section shows how you can simplify things.

Generic Methods

You can also use the Of keyword in the definition of a method. Consider the following procedure, which you can place inside a module:

```
' Exchange two arguments passed by address.
Public Sub Swap(Of T)(ByRef x As T, ByRef y As T)
   Dim tmp As T = x
   x = y
   y = tmp
End Sub
```

You can call the Swap method by passing two variables of the same type:

```
Dim n1 As Integer = 123
Dim n2 As Integer = 456
Swap(Of Integer)(n1, n2)
Console.WriteLine("n1={0}, n2={1}", n1, n2)    ' => n1=456, n2=123
```

It's remarkable that in most cases the Visual Basic compiler doesn't even require the Of keyword in the method invocation:

```
' The following statement works correctly.
Swap(n1, n2)
```

At times you do need to specify the Of clause when invoking a generic method. Consider the following definition:

```
Sub DoSomething(Of T)(ByVal x As T, ByVal y As T)
   …
End Sub
```

The following client code works correctly even if no Of keyword is used because the compiler can determine the generic parameter to be passed behind the scenes by looking at the type of the first argument passed to the method:

```
DoSomething(123, 456)            ' Same as DoSomething(Of Integer)
DoSomething(123.56, 456.78)      ' Same as DoSomething(Of Double)
```

However, you have a problem when the two arguments have a different type. For example, this code:

```
Dim l As Long = 456
Dim n As Integer = 123
DoSomething(l, n)
```

fails to compile with the following error message:

```
Type argument inference failed for type parameter 'T' of 'Public Sub DoSomething(Of T)
(x As T, y As T)'. Type argument inferred from the argument passed to parameter 'y'
conflicts with the type argument inferred from the argument passed to parameter 'x'.
```

This error message is a bit surprising because if the compiler looks at the first value passed to the method and infers that type T stands for Long, it should be able to automatically convert the second argument from Integer to Long. However, it is evident that in this case the Visual Basic compiler isn't able to perform even a widening conversion automatically.

You can get rid of the compilation error in two ways: either by manually converting the second argument to the same type as the first one or by specifying the Of clause in the method call:

```
' Both these statements work correctly.
DoSomething(l, CLng(n))
DoSomething(Of Long)(l, n)
```

Finally notice that only generic methods are supported; there is no such thing as a generic property, field, or event. In other words, Visual Basic refuses to compile this code:

```
Property Value(Of T)() As T
   …
End Property
```

However, you can have a property that reuses a generic parameter defined in the enclosing class:

```
Public Class Item(Of T)
   Property Value() As T
      …
   End Property
End Class
```

Setting the Default Value

One interesting detail about Visual Basic generics is that you can deal with reference types and value types in the same way. To see what I mean, let's extend the implementation of the ReadOnlyList class with the ability to clear all the elements that are currently stored in the collection:

```
' (Inside the ReadOnlyList class…)
Public Sub Reset()
   ' Reset all existing elements to the type's default value.
```

```
   For i As Integer = 0 To Me.Count - 1
      values(i) = Nothing
   Next
End Sub
```

The purpose of the Reset method is to assign the type's default value to each element; if the ReadOnlyList class stores strings or other kinds of objects, the default value is Nothing. However, if the ReadOnlyList class stores numbers or other value types, assigning the Nothing value should throw an exception at run time because you can't store Nothing in a value type variable, right?

Wrong. When Nothing is assigned to a variable typed after a generic parameter—as is the case of the values array in preceding code—the assignment is guaranteed not to fail even if the generic argument designates a value type. In this case, the default value for that type—that is, zero for numeric types, a null globally unique identifier (GUID) for the System.Guid type, and so forth—is assigned instead.

Even if it isn't immediately apparent, this feature enables you to test whether a given element matches the default value for the type. Here's an example:

```
Function IsDefaultValue(Of T)(ByVal value As T) As Boolean
   Dim defValue As T = Nothing
   Return Object.Equals(value, defValue)
End Function
```

Generic Interfaces

You can use generics with classes, structures, interfaces, and delegates (but not with modules and enum types). Generic structures work exactly the same way as generic classes do, but generic interfaces need some additional clarifications. The following code defines a generic interface and a class that implements that interface:

```
Interface IAdder(Of T)
   Function Add(ByVal n1 As T, ByVal n2 As T) As T
End Interface

Class Adder
   Implements IAdder(Of Integer)

   Public Function Add(ByVal n1 As Integer, ByVal n2 As Integer) As Integer _
         Implements IAdder(Of Integer).Add
      Return n1 + n2
   End Function
End Class
```

It is legal to implement multiple versions of the same generic interface, as in this code:

```
Class Adder
   Implements IAdder(Of Integer), IAdder(Of Double)

   Public Function Add(ByVal n1 As Integer, ByVal n2 As Integer) As Integer _
         Implements IAdder(Of Integer).Add
```

```
      Return n1 + n2
   End Function

   Public Function Add(ByVal n1 As Double, ByVal n2 As Double) As Double _
         Implements IAdder(Of Double).Add
      Return n1 + n2
   End Function
End Class
```

The most important difference between a standard interface and a generic interface is that the latter can avoid a box operation when method arguments are of a value type. For example, before generics were introduced the only way you could define a universal IAdder interface was to use Object arguments, as in the following:

```
Interface IAdder
   Function Add(ByVal n1 As Object, ByVal n2 As Object) As Object
End Interface
```

Implementing such an interface in a class would require that both the arguments and the return value—all of which are numbers, and therefore value types—be boxed and unboxed. By comparison, no boxing occurs when you implement the IAdder(Of T) interface if T is a value type.

The .NET Framework defines several generic interfaces, most of which are the generic version of weakly typed interfaces. These are the most important ones:

- IComparable(Of T), the strong-typed version of IComparable
- IComparer(Of T), the strong-typed version of IComparer
- IEquatable(Of T), which exposes an Equals method that takes an argument of a specific type
- IEnumerable(Of T) and IEnumerator(Of T), which allow a class to support For Each loops
- ICollection(Of T), which represents a collection of elements of type T
- IList(Of T), which represents a series of elements of type T
- IDictionary(Of K, V), which represents a dictionary of elements of type V indexed by keys of type K.

If the type that implements the interface is itself a generic type, the generic parameter can appear in the Of clause of the Implements statement, as in the following code:

```
Public Class TestComparer(Of T)
   Implements IComparer(Of T)
   Public Function Compare(ByVal x As T, ByVal y As T) As Integer _
      Implements IComparer(Of T).Compare
      …
   End Function
End Class
```

In practice, however, implementing generic interfaces in this way is difficult and sometimes impossible. For example, there is no simple way to implement correctly the Compare method in the previous code snippet because the code inside the method can't make any assumption about how two elements of type T can be compared to each other and can't use comparison operators with them. (You can sometimes work around this limitation by enforcing a constraint on the generic parameter, as you'll learn in a following section.)

Interestingly, many primitive .NET types have been expanded to implement the IEquatable(Of T) and IComparable(Of T) interfaces. For example, you can now invoke the strong-typed versions of the Equals and IComparable interfaces for all numeric types:

```
Sub TestInteger(ByVal value As Integer)
   ' These statements box their value in .NET Framework 1.1, but not in .NET Framework 2.0.
   If value.Equals(0) Then Console.WriteLine("It's zero")
   If value.CompareTo(0) > 0 Then Console.WriteLine("It's positive")
End Sub
```

You can't use the generic parameter as a direct argument of the Implements keyword. In other words, the following statements don't compile:

```
Public Class TestClass(Of T)
   Implements T
   …
End Class
```

Note A few generic interfaces inherit from the corresponding nongeneric one. For example, IEnumerable(Of T) inherits from IEnumerable; therefore, a class that implements the generic interface must implement both the IEnumerable(Of T).GetEnumerator method and the IEnumerator.GetEnumerator. Similarly, the IEnumerator(Of T) interface inherits from IEnumerator; therefore, a class that implements IEnumerator(Of T) must expose all three members of IEnumerator plus the strongly typed version of the Current property. (For more information about the IEnumerator interface, read Chapter 10, "Interfaces.")

Generics and Overloading

You can define generic types that have the same name but different numbers of generic parameters. For example, the following classes can coexist in the same namespace:

```
Public Class MyType
   …
End Class

Public Class MyType(Of T)
   …
End Class

Public Class MyType(Of T, K)
   …
End Class
```

This feature is similar to method overloading in the sense that the compiler chooses the type with the number of generic parameters that matches the number of generic arguments passed by the calling code:

```
Dim t1 As MyType                    ' An instance of the first class
Dim t2 As MyType(Of Long)           ' An instance of the second class
Dim t3 As MyType(Of Long, Double)   ' An instance of the third class
```

Similarly, you can define multiple methods with the same name and different sets of generic parameters. In this case, however, the rules are slightly more complicated and you must be aware of a few subtleties. Let's consider the following methods:

```
Sub DoTask(Of T, P)(ByVal x As T, ByVal y As P)
   Console.WriteLine("First version")
End Sub

Sub DoTask(Of T)(ByVal x As T, ByVal y As String)
   Console.WriteLine("Second version")
End Sub

Sub DoTask(Of T)(ByVal x As T, ByVal y As T)
   Console.WriteLine("Third version")
End Sub
```

In most cases, the Visual Basic compiler is smart enough to generate code that invokes the most specific version, even if you omit the Of clause in the method call:

```
DoTask(123, 456.78)                 ' Calls DoTask(Of Integer, Double)
DoTask(123, "abc")                  ' Calls DoTask(Of Integer, String)
```

However, if you attempt to pass two arguments of the same type, for example, two integers, the compiler complains and explains that overload resolution failed because no method is specific to the arguments being passed:

```
' *** Next statement raises a compilation error.
DoTask(123, 123)
```

To solve the problem you must give the compiler a hint about which version you want to be invoked:

```
' Next statement compiles correctly and invokes the third version of the method.
DoTask(Of Integer)(123, 123)
```

Generics and Inheritance

Earlier in this chapter, I stated that you can use a generic parameter anywhere in a class, as if it were a regular type name. Well, that description wasn't exactly accurate because a few exceptions exist:

- You can't use a generic parameter in the Inherits clause; in other words, you can't inherit from a type passed as a generic parameter.

- You can't use a generic parameter to reference an interface in the Implements clause, as I explain at the end of the section titled "Generic Interfaces" earlier in this chapter.
- You can't use a generic parameter in an attribute declaration.
- You can't use a generic parameter in a Declare statement or a method that is marked with the DllImport attribute, that is, in a method that runs unmanaged code.

The first limitation implies that you can't inherit a type from another type defined by means of a generic parameter:

```
' *** The following code doesn't compile.
Public Class TestClass(Of T)
   Inherits T
   …
End Class
```

Nothing prevents you, however, from using a generic type in the Inherits clause, which is in fact a rather common case. For example, the following two classes are based on the Relation type defined in an earlier section:

```
Public Class PersonCompanyRelation
   Inherits Relation(Of Person, Company)

   Public Sub New(ByVal person As Person, ByVal company As Company)
      MyBase.New(person, company)
   End Sub
End Class

Public Class PersonCompanyRelationList
   Inherits List(Of PersonCompanyRelation)
End Class
```

Thanks to these two classes, the client code that puts Person and Company objects in relation to each other can be simplified as follows:

```
Dim ca As New Company("Code Architects")
Dim john As New Person("John", "Evans")
Dim ann As New Person("Ann", "Beebe")

Dim relations As New PersonCompanyRelationList
Dim relJohnCa As New PersonCompanyRelation(john, ca)
relations.Add(relJohnCa)
relations.Add(New PersonCompanyRelation(ann, ca))
```

The GetEmployees method has a simpler and more readable declaration as well:

```
Function GetEmployees(ByVal relations As PersonCompanyRelationList, _
      ByVal company As Company) As List(Of Person)
   Dim result As New List(Of Person)
   For Each rel As Relation(Of Person, Company) In relations
      If rel.Object2 Is company Then result.Add(rel.Object1)
   Next
   Return result
End Function
```

The practice of defining and using a standard class that inherits from and wraps a generic type has several advantages:

- The structure and the syntax of client code are simpler.
- The client code can be written even in .NET languages that don't support generics, or even in unmanaged code.

Generics are fully CLS-compliant; therefore, all major .NET languages support them and you can freely expose them as parameters or return values of public methods. Even so, however, you might decide not to expose a generic type to the outside of your assembly to keep it fully interoperable with all .NET languages as well as with unmanaged clients.

Generics and the TypeOf . . . Is Operator

In general, a bound generic type can be used whenever you can use a regular type, as in this code:

```
If TypeOf obj Is List(Of String) Then
   Dim list As List(Of String) = DirectCast(obj, List(Of String))
End If
```

This rule holds true only for bound generic types, which represent real types, and isn't valid for open generic types, which represent a type definition rather than a real type. For example, the following code isn't valid (unless it appears inside a generic type that takes the T parameter):

```
' *** This code causes the following compile error: Type T is not defined.
If TypeOf obj Is List(Of T) Then
   ' obj is an open list type, such as List(Of Integer) or List(Of String).
End If
```

The previous test is rarely useful, because—even if it were a valid Visual Basic statement—you couldn't cast an object instance to a generic List(Of T) variable. As a matter of fact, you can't define such a variable:

```
' *** This statement causes two compile errors: Type T is not defined.
Dim list as List(Of T) = DirectCast(obj, List(Of T))
```

Let's see which options you have. If you simply must determine whether an object is an instance of a bound generic type, you can use this code:

```
If obj IsNot Nothing AndAlso obj.GetType().IsGenericType Then
   ' obj is an instance of a generic type.
End If
```

However, if you need to check whether an object is an instance of a generic bound type that derives from a given open generic definition, such as a type of the form List(Of T), you must use a different approach, one based on reflection. The FullName of a generic type definition consists of the complete name of the generic class, followed by an inverse quote character, and then the number of type parameters. For example, the full name of the List(Of T) generic type definition is this:

```
System.Collections.Generic.List`1
```

The FullName of a bound generic type is obtained by concatenating the complete name of the type arguments (enclosed between square brackets) to the previous string. For example, the full name of the List(Of Integer) type is

```
System.Collections.Generic.List`1[[System.Int32, mscorlib, Version=2.0.0.0,
   Culture=neutral, PublicKeyToken=b77a5c561934e089]]
```

Armed with this knowledge, you can test whether an object instance is a bound generic type of a List(Of T) using this code:

```
If obj IsNot Nothing AndAlso obj.GetType().FullName.StartsWith( _
      "System.Collections.Generic.List`1") Then
   ' obj is a generic type of the form List(Of T).
End If
```

However, notice that the previous test isn't perfectly equivalent to the TypeOf operator, which also tests whether the first argument is an instance of any type derived from the type specified in the second argument. If you must perform this sort of test, you must adopt a technique based on reflection:

```
' Test whether obj is a List(Of T) or derives from a List(Of T) type.
If obj IsNot Nothing Then
   Dim type As Type = obj.GetType()
   Do
      If type.FullName.StartsWith("System.Collections.Generic.List`1") Then
         Console.WriteLine("TypeOf obj Is List(Of T) is true!")
         Exit Do
      End If
      type = type.BaseType
   Loop Until type Is Nothing
End If
```

You can read more about reflection, the GetType operator, and the methods of the System.Type class in Chapter 18, "Reflection."

Testing and Converting Generic Values

A common problem with generics is that no evident way exists to convert a generic value into a more specific type. For example, consider this code:

```
Sub TestMethod(Of T)(ByVal value As T)
   If TypeOf value Is Integer Then       ' *** Compilation error
      Dim n As Integer = CInt(value)     ' *** Compilation error
      …
   End If
End Sub
```

The comments highlight the two statements that cause a compilation error. The first error occurs because the first argument of TypeOf must be a reference type, but the compiler has no clue about the generic T type; the second error occurs because the compiler knows nothing about the T type and can't guarantee that the CInt operator can convert an instance of T into an integer.

In cases like this, the simplest solution is to convert the argument to Object and then deal with it as you would normally:

```
Sub CheckArguments(Of T)(ByVal value As T)
   Dim obj As Object = CObj(value)
   If TypeOf obj Is Integer Then
      Dim n As Integer = CInt(obj)
      …
   End If
End Sub
```

The problem with this approach is that it causes the value to be boxed if T is a value type. If you don't need to extract its value and want simply to test its type, you can save a box operation by means of a reflection-based technique:

```
   If value IsNot Nothing AndAlso value.GetType() Is GetType(Integer) Then
      …
   End If
```

Generic Constraints

Consider the following generic method, which returns the highest value among its arguments:

```
Public Function Max(Of T)(ByVal ParamArray values() As T) As T
   Dim result As T = values(0)
   For i As Integer = 1 To UBound(values)
      ' *** The next statement causes the following compilation error:
      '     "Operator '>' is not defined for types 'T' and 'T'."
      If values(i) > result Then result = values(i)
   Next
   Return result
End Function
```

As the remark in the preceding code indicates, the greater than sign (>) causes a compilation error because the compiler can't be sure that client code calls the method only with arguments that support this operator. This problem occurs quite frequently when you are working with generics, but you can work around it by enforcing a constraint for the T type. For example, you can require that the method be called only with types that support the IComparable interface:

```
Public Function Max(Of T As IComparable)(ByVal ParamArray values() As T) As T
   …
End Function
```

Because T surely exposes the IComparable interface, the code in the method can safely invoke the CompareTo method to calculate the highest value in the array:

```
Public Function Max(Of T As IComparable)(ByVal ParamArray values() As T) As T
   Dim result As T = values(0)
   For i As Integer = 1 To UBound(values)
      If result.CompareTo(values(i)) < 0 Then result = values(i)
   Next
   Return result
End Function
```

Here's a piece of client code that uses the Max function:

```
' No need to specify the Of clause in calling the method.
Console.WriteLine(Max(12, 23, 6, -1))                    ' => 23
```

Visual Basic 2005 supports five types of constraints:

- **Interface constraint** The type argument must implement the specified interface.
- **Inheritance constraint** The type argument must derive from the specified base class.
- **Class constraint** The type argument must be a reference type.
- **Structure constraint** The type argument must be a value type.
- **New constraint** The type argument must expose a public parameterless (default) constructor.

Notice that you can't define a constraint specifying that a type must expose a constructor with a given signature; the New constraint ensures that one of the public constructors of the type has no arguments.

You can read more about these constraint types in the following sections.

The Interface Constraint

This kind of constraint is often used with the IComparable interface, as in the code example just shown. For instance, here's an interesting recursive method that returns the median value in a list. (The median of a list of *N* elements is the value that is greater than *N*/2 elements and less than the remaining *N*/2 elements.)

```
Function MedianValue(Of T As IComparable)(ByVal list As List(Of T), _
      Optional ByVal position As Integer = -1) As T
   ' Provide a default value for second argument.
   If position < 0 Then position = list.Count \ 2

   ' If the list has just one element, we've found its median.
   Dim guess As T = list(0)
   If list.Count = 1 Then Return guess
   ' This list will contain values lower and higher than the current guess.
   Dim lowerList As New List(Of T)
   Dim higherList As New List(Of T)

   For i As Integer = 1 To list.Count - 1
      Dim value As T = list(i)
      If guess.CompareTo(value) <= 0 Then
         ' The value is higher than or equal to the current guess.
         higherList.Add(value)
      Else
         ' The value is lower than the current guess.
         lowerList.Add(value)
      End If
   Next
```

```
   If lowerList.Count > position Then
      ' The median value must be in the lower-than list.
      Return MedianValue(lowerList, position)
   ElseIf lowerList.Count < position Then
      ' The median value must be in the higher-than list.
      Return MedianValue(higherList, position - lowerList.Count - 1)
   Else
      ' The guess is correct.
      Return guess
   End If
End Function
```

Of course, you can evaluate the median value of an array by sorting the array and then picking the element at index *N*/2, but MedianValue is typically faster because it saves you the sort step.

You can retrieve other interesting values in a list by passing a second argument to the MedianValue method. For example, by passing the value 0, the method returns the lowest value in the list; by passing the value 1, the method returns the second lowest value in the list; by passing the value *N* − 1, the method returns the highest value in a list of *N* elements; by passing the value *N* − 2, the method returns the second highest value in the list, and so forth.

You can specify a generic interface as a constraint. For example, you can improve the MedianValue as follows:

```
Function MedianValue(Of T As IComparable(Of T))(ByVal list As List(Of T), _
   Optional ByVal position As Integer = -1) As T
   …
End Function
```

The advantage of using a generic interface instead of a weakly typed interface is that no boxing occurs when the new version of the MedianValue method invokes the CompareTo method of the interface:

```
' In the new version of MedianValue, this statement causes no boxing.
If guess.CompareTo(value) <= 0 Then
```

All numeric types in the .NET Framework implement the IComparable(Of T) and IEquatable(Of T) interfaces; thus, the new version of the MedianValue method can work with all the integer and floating-point numeric types. If you define a new numeric data type, it is strongly recommended that you implement the IComparable(Of T) and IEquatable(Of T) generic interfaces.

You can use the interface constraint with any interface, not just IComparable. For example, a constraint for the ISerializable interface ensures that the generic type or method can be used only with types that can be serialized and deserialized from a file or a database field. (Read Chapter 21, "Serialization," for more information about the ISerializable interface.) In the remainder of this chapter, I provide other examples of interface constraints.

The Inheritance Constraint

The inheritance constraint tells the Visual Basic compiler that a generic argument can only be a type that derives from the specified class. The syntax is similar to the interface constraint:

```
' This generic class can be used only with types that derive
' from System.Windows.Forms.Control.
Public Class ControlCollection(Of T As System.Windows.Forms.Control)
   Inherits List(Of T)
   …
End Class
```

Because of the inheritance constraint, you can use the ControlCollection class to create a collection of Button or TextBox controls, but not Person or Company objects. In addition to improved robustness, the inheritance constraint gives you the ability to invoke any Public member of the type specified by the constraint. For example, the code in the ControlCollection class can safely access members of the Control type, such as the Text and ForeColor properties. Unfortunately, the presence of the inheritance constraint doesn't suffice to enable you to invoke the constructor of the class because classes that derive from the same base type can define a different set of constructors and even have no constructors at all. (See the section titled "The New Constraint" later in this chapter for more details.)

A few generics defined in the .NET Framework use the inheritance constraint. For example, the System.EventHandler(Of T) generic type is a delegate that can be used to define an event and mandates that the T type inherits from System.EventArgs. If EventHandler(Of T) were defined in Visual Basic, it would look like this:

```
Public Delegate Sub EventHandler(Of T As EventArgs)(ByVal sender As Object, ByVal e As T)
```

(You can see this type in action in the section titled "Generics and Events" later in this chapter.) There are a few restrictions for the type that follows the As clause in an inheritance constraint. For obvious reasons, the type can't be sealed (NotInheritable in Visual Basic) and therefore it can't be a structure. Also, you can't use the System.Object, System.ValueType, or System.Delegate types or any delegate type.

The Class and Structure Constraints

A generic parameter can be followed by the As Class clause, to specify that the type parameter is a reference type, or by the As Structure clause, to indicate that the type parameter is a value type:

```
Public Class ObjectCollection(Of T As Class)
   …
End Class

Public Class ValueCollection(Of T As Structure)
   …
End Class
```

Note In theory you might have two generic types with the same name that differ only by the Class or Structure constraint applied to their generic argument because the Visual Basic compiler should be able to use one or the other depending on whether the generic argument is a class or a structure. However, the compiler isn't that smart, and the general rule still applies: a namespace can contain two generic types with the same name only if they take a different number of generic parameters.

The class constraint (but not the structure constraint) adds the ability to use the Is, IsNot, and TypeOf . . . Is operators. For example, if you apply this constraint to the type parameters of the Relation generic class, you can define a Contains method that uses the Is operator to check whether a given object is part of the relation:

```
Public Class Relation(Of T1 As Class, T2 As Class)
   Public ReadOnly Object1 As T1
   Public ReadOnly Object2 As T2

   Public Sub New(ByVal obj1 As T1, ByVal obj2 As T2)
      Me.Object1 = obj1
      Me.Object2 = obj2
   End Sub

   Public Function Contains(ByVal obj As Object) As Boolean
      Return Me.Object1 Is obj OrElse Me.Object2 Is obj
   End Function
End Class
```

Notice that you must define the Contains method so that it takes a generic Object argument. In this particular case, it doesn't really affect the quality of your code because the two objects passed to the constructor of the Relation class are reference types and therefore no box operation occurs when the Contains method is used appropriately (unless you mistakenly pass it a value-typed element that isn't part of the relation). You might believe that you can enforce a more robust code by offering two overloads for the Contains method, as in the following:

```
   Public Function Contains(ByVal obj As T1) As Boolean
      Return Me.Object1 Is obj
   End Function
   Public Function Contains(ByVal obj As T2) As Boolean
      Return Me.Object1 Is obj
   End Function
```

This code compiles correctly, but only as long as the client code never creates a Relation object whose two generic parameters are the same type. For example, the following code doesn't compile:

```
Dim john As New Person("John", "Evans")
Dim ann As New Person("Ann", "Beebe")
Dim rel As New Relation(Of Person, Person)(john, ann)
' Next statement raises the following compilation error: "Overload resolution
' failed because no accessible 'Contains' is most specific for these arguments…"
Dim found As Boolean = rel.Contains(john)
```

Here's what has happened: when the compiler replaces both T1 and T2 with the Person type, it finds that two Contains methods are using the same signature. Oddly, the compiler should flag the statement that creates the Relation object as an error because the resulting bound generic class contains two overloaded methods with the same signature. Instead, the error is emitted only if the project actually contains a call to that method. Mysteries of .NET generics ...

The New Constraint

The New constraint adds the requirement that the type passed as the generic parameter has a public parameterless constructor. This constraint allows you to create instances of the specified type, so you often use it in factory methods such as the following:

```
Public Function CreateObject(Of T As New) As T
   Return New T
End Function
```

A better example shows how you can initialize an array of objects of a given type:

```
Public Function CreateArray(Of T As New)(ByVal numEls As Integer) As T()
   Dim values(numEls - 1) As T
   For i As Integer = 0 To numEls - 1
      values(i) = New T
   Next
   Return values
End Function
```

The New constraint is often used in conjunction with other constraints, as explained in the followed section.

Multiple Constraints

It is possible to enforce more than one constraint by enclosing the constraints in a pair of braces. This syntax is especially useful to combine the New constraint with the interface constraint or the inheritance constraint, or to enforce multiple interface constraints on the same generic parameter, as in this code:

```
Public Class Widget(Of T As {New, IComparable}, V As {IComparable, IConvertible})
   …
End Class
```

The following example uses a compound constraint to implement a generic type that behaves like a sortable array:

```
Public Class SortableArray(Of T, C As {New, IComparer(Of T)})
   Dim values() As T

   Public Sub New(ByVal highestIndex As Integer)
      ReDim values(highestIndex)
   End Sub

   Public Sub Sort()
      ' Sort the array using the specified comparer object.
```

```
      Array.Sort(values, New C)
   End Sub

   Default Public Property Item(ByVal index As Integer) As T
      Get
         Return values(index)
      End Get
      Set(ByVal value As T)
         values(index) = value
      End Set
   End Property
End Class
```

To see the SortableArray class in action you must define a suitable comparer class, which can be as simple as this one:

```
Public Class ReverseIntegerComparer
   Implements IComparer(Of Integer)

   Public Function Compare(ByVal x As Integer, ByVal y As Integer) As Integer _
          Implements IComparer(Of Integer).Compare
      ' Return -1 if x > y, +1 if x < y, 0 if x = y.
      Return Math.Sign(y - x)
   End Function
End Class
```

Finally, you can define a SortableArray object that contains integers and that, when sorted, arranges elements in reverse order:

```
' A sortable array that can contain 11 elements
Dim arr As New SortableArray(Of Integer, ReverseIntegerComparer)(10)
arr(0) = 123
…
' Sort the array (in reverse order).
arr.Sort()
```

Note You might wonder why the Compare method uses a Math.Sign function instead of a simpler call to the CompareTo method, exposed by the IComparable interface:

```
Return DirectCast(y, IComparable).CompareTo(x)
```

The reason is subtle and has to do with performance. The previous statement, in fact, causes two hidden box operations: first, the *y* variable is boxed when it is cast to the IComparable interface; second, the *x* Integer value is passed to an Object argument and therefore must be boxed as well. You can avoid the second box operation by casting to the IComparable(Of Integer) interface, as in this code:

```
Return DirectCast(y, IComparable(Of Integer)).CompareTo(x)
```

However, you can't avoid the first box operation, caused by the DirectCast operator, which in turn is necessary because the CompareTo method is private and can be accessed only through the IComparable interface.

In this particular case you can improve performance by passing the $y - x$ difference to the Math.Sign method; when you have no other solution but to use DirectCast to invoke a private interface member, you can't avoid the extra box operation.

Checking a Constraint at Run Time

As sophisticated as it is, the constraint mechanism isn't perfect. For example, it isn't possible to request that a type passed as an argument implements *either* interface A or interface B (or both), or that it *doesn't* implement an interface or inherit from a given base class, or that it is marked with a given attribute, or that it exposes a method or a constructor with a given name and signature. And you can't check that *at least* one of the type arguments (but not necessarily all of them) implement a given interface. In cases like these, you can't specify a standard constraint; instead, the best you can do is check the condition at run time.

Provided that you know how to test the condition, it's easy to check the constraint in the generic type's constructor, as in this case:

```
Public Class ClassWithRuntimeConstraint(Of T)
   Sub New()
      ' Check that the T type implements either IDisposable or ICloneable.
      ' (We need reflection to perform this test.)
      If Not GetType(IDisposable).IsAssignableFrom(GetType(T)) AndAlso _
         Not GetType(ICloneable).IsAssignableFrom(GetType(T)) Then
         Throw New ArgumentException("Invalid type argument")
      End If
      ' Continue here with the constructor…
   End Sub
End Class
```

Although this approach works, it is less than optimal because the condition is checked each time an instance of the TestClass type is created. A better approach is to place the condition in the static constructor of the type, which is executed only once during the application's lifetime:

```
   ' Check type constraint in the static type constructor.
   Shared Sub New()
      If Not GetType(IDisposable).IsAssignableFrom(GetType(T)) AndAlso _
         Not GetType(ICloneable).IsAssignableFrom(GetType(T)) Then
         Throw New ArgumentException("Invalid type argument")
      End If
   End Sub
```

Advanced Topics

Generic types are new, powerful tools in the hands of expert developers. As with all power tools, it takes some time to master them.

Nullable Types

Virtually all databases support the concept of nullable columns, namely, columns that can contain the special NULL value. Such a special value is often used as an alias for "unknown value" or "unassigned value." The use of nullable columns tends to make database-oriented applications more complicated than they need to be. For example, you can't move a value from a nullable numeric column into an Int32 or Double .NET variable without testing the

value against the DBNull.Value special value. (The actual method or operation you must perform depends on the ADO.NET object you're using.)

Microsoft .NET Framework version 2.0 introduces the concept of nullable types, that is, value types that can be assigned a special null value. Notice that only value types need to be treated in this way because reference types—such as strings and arrays—can use the Nothing value as an alias for the null state.

As you probably have already guessed by now, .NET nullable types are based on generics. For example, here's how you can define a nullable Integer value:

```
' Declare an "unassigned" nullable value.
Dim n As Nullable(Of Integer)
' Assign it a value.
n = 123
' Reset it to the "unassigned" state.
n = Nothing

' You can declare and assign a nullable value in these two ways.
Dim d1 As Nullable(Of Double) = 123.45
Dim d2 As New Nullable(Of Double)(123.45)
```

The Nullable(Of T) generic type exposes two key properties, both of which are read-only. The HasValue property returns False if the element is in the unassigned state; the Value property returns the actual value if HasValue is True; otherwise, it throws an InvalidOperationException object:

```
If n.HasValue Then
   Console.WriteLine("Value is {0}.", n.Value)
Else
   Console.WriteLine("No value has been assigned yet.")
End If
```

The Nullable(Of T) type supports conversions to and from the T type. For example, you can convert a Double value to a Nullable(Of Double) value and vice versa, but the latter conversion fails if the nullable element has no value; therefore, it is considered a narrowing conversion and requires an explicit CType operator (or equivalent, such as CInt or CDbl):

```
Dim value As Double = 123.45
' This conversion can never fail.
Dim value2 As Nullable(Of Double) = value
' The conversion in the opposite direction can fail; thus, it must be explicit.
Dim value3 As Double = CDbl(value2)
```

Even though nullable types appear to be structures, they are given special treatment at the IL level and are often interchangeable with the underlying type they can contain. This special support becomes apparent in the way nullable values are boxed and unboxed. Consider this code:

```
' Create a null value and box it.
Dim n As New Nullable(Of Integer)
Dim obj As Object = n
' obj contains something, yet next statement displays True.
```

```
Console.WriteLine(obj Is Nothing)         '  => True
' You can unbox obj to a Nullable object or directly to an Integer value.
' If the Nullable object has no value, the target variable is assigned the default value.
Dim n2 As Integer = CInt(obj)             ' n2 is assigned 0.
```

Even though you can use a nullable type in most of the places where the corresponding non-nullable type can appear, you have to account for one weird limitation: you can't pass a nullable type as a generic argument that has a structure constraint. In other words, assume you have the following generic class:

```
Public Class TestClass(Of T As Structure)
   …
End Class
```

If you now attempt to pass a nullable type to the T argument, as in this code:

```
Dim o As TestClass(Of Nullable(Of Integer))
```

you get the following error message:

```
'System.Nullable' does not satisfy the 'Structure' constraint for type
parameter 'T'. Only non-nullable 'Structure' types are allowed.
```

Math and Comparison Operators

Unfortunately, the Nullable(Of T) generic type supports neither math nor comparison operators. In other words, you can't directly add two nullable types. Instead, you must first convert them explicitly to the corresponding numeric type:

```
' This code assumes that d1 and d2 are Nullable(Of Double) elements.
If d1.HasValue AndAlso d2.HasValue Then
   Dim sum As Double = d1.Value + d2.Value
End If
```

Another solution for this issue is based on the GetValueOrDefault method, which returns either the current value (if HasValue is True) or the default value:

```
' Add to nullable numbers, using zero if the value is null.
Dim sum As Double = d1.GetValueOrDefault() + d2.GetValueOrDefault()
```

The GetValueOrDefault method can take one argument, which is used as the default value if HasValue is False:

```
' Assign the current value, or negative infinity if value is null.
Dim value As Double = d1.GetValueOrDefault(Double.NegativeInfinity)
```

You can check whether two nullable values are equal by using the Equals(Of T) method, which nullable types inherit from the IEquatable generic interface. This feature compensates for the lack of support of the equal (=) and not equal (<>) operators:

```
If d1.Equals(0) OrElse d1.Equals(d2) Then
   ' d1 is either zero or is equal to d2.
```

```
ElseIf d1.Equals(Nothing) Then
   ' This is another way to test whether a nullable type has a value.
End If
```

Alternatively, you can use the Nullable.Equals(Of T) static method:

```
If Nullable.Equals(d1, d2) Then
   ' d1 is equal to d2.
End If
```

You can also compare two nullable values by means of the Nullable.Compare static method; according to this method, a null value is always less than any nonnull value:

```
Select Case Nullable.Compare(d1, d2)
   Case -1
      Console.WriteLine("d1 is null or is less than d2")
   Case 1
      Console.WriteLine("d2 is null or is less than d1")
   Case 0
      Console.WriteLine("d1 and d2 have same value or are both null.")End Select
```

Three-Valued Boolean Logic

Three-value logic is quite common when you are dealing with Boolean expressions with operands that can take the True, False, or Unknown value. For example, SQL makes extensive use of three-value logic because it must account for nullable fields. Consider the following SQL statement:

```
SELECT * FROM Customers WHERE City="Rome" Or Country="Vatican"
```

If the City field is NULL, the City="Rome" subexpression is also NULL; however, if Country is equal to "Vatican," the second operand of the Or operator is True, which makes the entire WHERE clause True. In other words, a True operand makes the entire Or expression equal to True even if the other operand is NULL. Likewise, a False operand in an And expression makes the entire expression False, regardless of whether the other operand is known.

Given the similarities of three-value logic values with nullable types you might believe that you can implement the former ones by using the Nullable(Of Boolean) type as a base class and then overload the And, Or, Not, and Xor operators (and a few others). Unfortunately, this isn't a viable solution because the Nullable(Of T) type is a structure and can't be the base class for another type. This means that the only way you can implement three-value logic in your application is by defining a new type from scratch, as in the following code:

```
Public Structure NullableBoolean
   Private m_HasValue As Boolean
   Private m_Value As Boolean

   Sub New(ByVal value As Boolean)
      m_HasValue = True
      m_Value = value
   End Sub
```

```
Public ReadOnly Property HasValue() As Boolean
   Get
      Return m_HasValue
   End Get
End Property

Public ReadOnly Property Value() As Boolean
   Get
      If Not m_HasValue Then Throw New InvalidOperationException()
      Return m_Value
   End Get
End Property

Public Overrides Function ToString() As String
   If Me.HasValue Then
      Return Me.Value.ToString()
   Else
      Return "Null"
   End If
End Function

Public Shared Operator =(ByVal v1 As NullableBoolean, _
      ByVal v2 As NullableBoolean) As NullableBoolean
   If v1.HasValue AndAlso v2.HasValue Then
      Return New NullableBoolean(v1.Value = v2.Value)
   Else
      Return New NullableBoolean
   End If
End Operator

Public Shared Operator <>(ByVal v1 As NullableBoolean, _
      ByVal v2 As NullableBoolean) As NullableBoolean
   Return Not (v1 = v2)
End Operator

Public Shared Operator And(ByVal v1 As NullableBoolean, _
      ByVal v2 As NullableBoolean) As NullableBoolean
   If (v1.HasValue AndAlso v1.Value = False) OrElse _
         (v2.HasValue AndAlso v2.Value = False) Then
      Return New NullableBoolean(False)
   ElseIf v1.HasValue AndAlso v2.HasValue Then
      Return New NullableBoolean(True)
   Else
      Return New NullableBoolean()
   End If
End Operator

Public Shared Operator Or(ByVal v1 As NullableBoolean, _
      ByVal v2 As NullableBoolean) As NullableBoolean
   If (v1.HasValue AndAlso v1.Value) OrElse _
         (v2.HasValue AndAlso v2.Value) Then
      Return New NullableBoolean(True)
   ElseIf v1.HasValue AndAlso v2.HasValue Then
      Return New NullableBoolean(False)
   Else
      Return New NullableBoolean()
```

```
      End If
   End Operator

   Public Shared Operator Not(ByVal v As NullableBoolean) As NullableBoolean
      If v.HasValue Then
         Return New NullableBoolean(Not v.Value)
      Else
         Return New NullableBoolean
      End If
   End Operator

   Public Shared Operator Xor(ByVal v1 As NullableBoolean, _
         ByVal v2 As NullableBoolean) As NullableBoolean
      If v1.HasValue AndAlso v2.HasValue Then
         Return v1.Value Xor v2.Value
      Else
         Return New NullableBoolean
      End If
   End Operator

   Public Shared Operator IsTrue(ByVal v As NullableBoolean) As Boolean
      Return v.HasValue AndAlso v.Value
   End Operator

   Public Shared Operator IsFalse(ByVal v As NullableBoolean) As Boolean
      Return v.HasValue AndAlso v.Value = False
   End Operator

   Public Shared Widening Operator CType(ByVal v As Boolean) As NullableBoolean
      Return New NullableBoolean(v)
   End Operator

   Public Shared Narrowing Operator CType(ByVal v As NullableBoolean) As Boolean
      If v.HasValue Then
         Return v.Value
      Else
         Throw New InvalidOperationException("Nullable objects must have a value")
      End If
   End Operator
End Structure
```

You use the NullableBoolean type as you'd use a Nullable(Of Boolean) type, except it supports all the operators you need when working with three-value logic:

```
Dim fal As NullableBoolean = False
Dim tru As NullableBoolean = True
Dim unk As NullableBoolean            ' Null is the default state.

Console.WriteLine(fal And unk)        ' => False
Console.WriteLine(tru And unk)        ' => Null
Console.WriteLine(fal Or unk)         ' => Null
Console.WriteLine(tru Or unk)         ' => True
Console.WriteLine(fal Xor unk)        ' => Null
Console.WriteLine(tru Xor unk)        ' => Null
Console.WriteLine(fal = unk)          ' => Null
Console.WriteLine(tru <> unk)         ' => Null
```

The CType operator allows you to convert a NullableBoolean to a Boolean by means of an explicit conversion:

```
' Throws an exception if the NullableBoolean element has an unknown value.
Dim ok As Boolean = CBool(fal)
```

The IsTrue and IsFalse operators add support for the AndAlso and OrElse keywords:

```
If fal AndAlso tru Then
   ' This block isn't executed.
End If
```

Support for Math Operators

As I emphasized many times in previous sections, a generic type can't perform any math operation on objects with a type defined by using a generic parameter. In general, no operator can be used and no method can be invoked on such objects. (As a special case, you can work around the lack of support of relational operators by enforcing a constraint for either the IComparable or the IEquatable interfaces.)

In a perfect world, all .NET numeric types would support a common interface that would allow a generic type to perform math. For example, suppose that the following interface were defined in the .NET Framework:

```
Public Interface IMath(Of T)
   Function Add(ByVal n As T) As T
   Function Subtract(ByVal n As T) As T
   Function Multiply(ByVal n As T) As T
   Function Divide(ByVal n As T) As T
End Interface
```

If all the .NET Framework numeric types supported the IMath(Of T) interface—in much the same way they support the IComparable(Of T) interface—a generic type could perform the four math operations on these types with no effort. Alas, this interface is neither defined in the .NET Framework nor implemented by any .NET type, so this approach isn't viable. It's a pity, and we can only hope that Microsoft will remedy this in a future version of the .NET Framework.

To understand how you can work around this issue, consider the relation between the IComparable(Of T) and the IComparer(Of T) interfaces. If you want to compare two objects that support the IComparable(Of T) interface, you can just invoke the CompareTo method that these objects expose. However, if the objects don't expose this interface, you can define a type that supports the IComparer(Of T) interface and that is capable of comparing two objects of type T.

Along the same lines, you can work around the lack of support for math operators by defining an ICalculator(Of T) interface, and then create one or more types that implement this interface;

these types provide the ability to perform math on elements of type T. Here's the definition of the ICalculator(Of T) interface:

```
Public Interface ICalculator(Of T)
   Function Add(ByVal n1 As T, ByVal n2 As T) As T
   Function Subtract(ByVal n1 As T, ByVal n2 As T) As T
   Function Multiply(ByVal n1 As T, ByVal n2 As T) As T
   Function Divide(ByVal n1 As T, ByVal n2 As T) As T
   Function ConvertTo(ByVal n As Object) As T
End Interface
```

Next, you need to implement one or more classes that implement this interface for all the numeric types in the .NET Framework and, optionally, for any custom type in your application that supports the four operators. You can adopt two strategies: you can have one separate class for each numeric type or an individual class that implements several versions of the interface, one of each numeric type you want to support.

The following NumericCalculator class implements the ICalculator interface for the Integer and the Double types, but you can easily extend it to support all other primitive .NET numeric types. As you can see, it's a lot of code, but it's mostly a copy-and-paste job:

```
Public Class NumericCalculator
   Implements ICalculator(Of Integer)
   Implements ICalculator(Of Double)

   ' The ICalculator(Of Integer) interface
   Public Function AddInt32(ByVal n1 As Integer, ByVal n2 As Integer) As Integer _
         Implements ICalculator(Of Integer).Add
      Return n1 + n2
   End Function
   Public Function SubtractInt32(ByVal n1 As Integer, ByVal n2 As Integer) As Integer _
         Implements ICalculator(Of Integer).Subtract
      Return n1 - n2
   End Function
   Public Function MultiplyInt32(ByVal n1 As Integer, ByVal n2 As Integer) As Integer _
         Implements ICalculator(Of Integer).Multiply
      Return n1 * n2
   End Function
   Public Function DivideInt32(ByVal n1 As Integer, ByVal n2 As Integer) As Integer _
         Implements ICalculator(Of Integer).Divide
      Return n1 \ n2
   End Function
   Public Function ConvertToInt32(ByVal n As Object) As Integer _
         Implements ICalculator(Of Integer).ConvertTo
      Return CInt(n)
   End Function

   ' The ICalculator(Of Double) interface
   Public Function AddDouble(ByVal n1 As Double, ByVal n2 As Double) As Double _
         Implements ICalculator(Of Double).Add
      Return n1 + n2
   End Function
   Public Function SubtractDouble(ByVal n1 As Double, ByVal n2 As Double) As Double _
         Implements ICalculator(Of Double).Subtract
```

```
      Return n1 - n2
   End Function
   Public Function MultiplyDouble(ByVal n1 As Double, ByVal n2 As Double) As Double _
         Implements ICalculator(Of Double).Multiply
      Return n1 * n2
   End Function
   Public Function DivideDouble(ByVal n1 As Double, ByVal n2 As Double) As Double _
         Implements ICalculator(Of Double).Divide
      Return n1 / n2
   End Function
   Public Function ConvertToDouble(ByVal n As Object) As Double _
         Implements ICalculator(Of Double).ConvertTo
      Return CDbl(n)
   End Function
End Class
```

Let's now see how you can leverage the NumericCalculator class in a generic type that works as a list but is also capable of performing some basic statistical operations on its elements:

```
Public Class StatsList(Of T, C As {New, ICalculator(Of T)})
   Inherits List(Of T)

   ' The object used as a calculator
   Dim calc As New C

   ' Return the sum of all elements.
   Public Function Sum() As T
      Dim result As T
      For Each elem As T In Me
         result = calc.Add(result, elem)
      Next
      Return result
   End Function

   ' Return the average of all elements.
   Public Function Avg() As T
      Return calc.Divide(Me.Sum, calc.ConvertTo(Me.Count))
   End Function
End Class
```

Using the StatsList generic type is a breeze:

```
Dim sl As New StatsList(Of Double, NumericCalculator)
For i As Integer = 0 To 10
   sl.Add(i)
Next
Console.WriteLine("Sum = {0}", sl.Sum)         ' => Sum = 55
Console.WriteLine("Average = {0}", sl.Avg)     ' => Average = 5
```

Generics and Events

Generics can greatly simplify the structure of types that contain public events. As you might recall from Chapter 7, "Delegates and Events," all event handlers must receive two arguments: *sender* and *e*, where the latter is a System.EventArgs (if the event doesn't expose any additional

property to subscribers) or an object that derives from System.EventArgs. To follow Microsoft guidelines closely, for each event that carries one or more arguments, you should define a type named *EventName*EventArgs that derives from EventArgs, the corresponding *EventName*-EventHandler delegate, and an On*EventName* overridable procedure that raises the event. It's a lot of work for just one event, and it's no surprise that most developers don't feel like writing all this code just to implement one event.

To see how the inheritance constraint can help you in streamlining the structure of events, let's suppose you are authoring an Employee class that exposes the Name and BirthDate properties and raises a *PropertyName*Changing event before either property is modified (so that subscribers can cancel the assignment) and a *PropertyName*Changed event after the property has been assigned. According to guidelines, you should define a class named NameChangingEventArgs that exposes the ProposedValue read-only string property (the value about to be assigned to the Name property) and the Cancel read-write Boolean property (which can be set to True by event subscribers to cancel the assignment). Likewise, you should define a class named BirthDateChangingEventArgs class, which exposes the same properties except that the ProposedValue property returns a Date value. Instead of defining two distinct classes, let's create a generic type named PropertyChangingEventArgs:

```
Public Class PropertyChangingEventArgs(Of T)
   ' Inheriting from CancelEventArgs adds support for the Cancel property.
   Inherits System.ComponentModel.CancelEventArgs

   Public Sub New(ByVal proposedValue As T)
      M_ProposedValue = proposedValue
   End Sub

   Private m_ProposedValue As T

   Public ReadOnly Property ProposedValue() As T
      Get
         Return m_ProposedValue
      End Get
   End Property
End Class
```

You now have two options. First, you can use the PropertyChangingEventArgs(Of String) type for the NameChanging event and the PropertyChangingEventArgs(Of Date) type for the BirthDateChanging event; in this case you'd need to edit the code slightly in the Employee class to account for these different names. Second, you can define two regular classes that inherit from the PropertyChangingEventArgs generic type:

```
Public Class NameChangingEventArgs
   Inherits PropertyChangingEventArgs(Of String)
   …
End Class

Public Class BirthDateChangingEventArgs
   Inherits PropertyChangingEventArgs(Of Date)
   …
End Class
```

In the remainder of this section, I assume that you've adopted the first approach and that all events are directly defined in terms of the PropertyChangingEventArgs(Of T) generic type.

The System.EventHandler(Of T) type is a generic delegate that can be passed any type that derives from System.EventArgs and that relieves you from defining a different delegate for each event. Thanks to this generic type and the nongeneric EventHandler type, you can define the four events in the Employee class as follows:

```
Public Class Employee
   Event NameChanging As EventHandler(Of PropertyChangingEventArgs(Of String))
   Event BirthDateChanging As EventHandler(Of PropertyChangingEventArgs(Of Date))
   Event NameChanged As EventHandler
   Event BirthDateChanged As EventHandler
   …
```

Adding support for the Name and BirthDate properties, and corresponding *Xxxx*Changing and *Xxxx*Changed events, is now straightforward:

```
   ' (Continuing previous code snippet…)
   Private m_Name As String

   Public Property Name() As String
      Get
         Return m_Name
      End Get
      Set(ByVal value As String)
         If m_Name <> value Then
            Dim e As New PropertyChangingEventArgs(Of String)(value)
            OnNameChanging(e)
            If e.Cancel Then Exit Property
            m_Name = value
            OnNameChanged(EventArgs.Empty)
         End If
      End Set
   End Property

   Private m_BirthDate As Date

   Public Property BirthDate() As Date
      Get
         Return m_BirthDate
      End Get
      Set(ByVal value As Date)
         If m_BirthDate <> value Then
            Dim e As New PropertyChangingEventArgs(Of Date)(value)
            OnBirthDateChanging(e)
            If e.Cancel Then Exit Property
            m_BirthDate = value
            OnBirthDateChanged(EventArgs.Empty)
         End If
      End Set
   End Property

   ' Protected OnXxxx methods
   Protected Overridable Sub OnNameChanging(ByVal e As _
```

```
         PropertyChangingEventArgs(Of String))
      RaiseEvent NameChanging(Me, e)
   End Sub

   Protected Overridable Sub OnNameChanged(ByVal e As EventArgs)
      RaiseEvent NameChanged(Me, e)
   End Sub

   Protected Overridable Sub OnBirthDateChanging(ByVal e As _
         PropertyChangingEventArgs(Of Date))
      RaiseEvent BirthDateChanging(Me, e)
   End Sub

   Protected Overridable Sub OnBirthDateChanged(ByVal e As EventArgs)
      RaiseEvent BirthDateChanged(Me, e)
   End Sub
End Class
```

Generics can help you reduce the amount of code needed to support events in one more way. The Set blocks in the Name and BirthDate property procedures are almost identical, except for the name of the EventArgs-derived class and the On*Xxxx* methods. Even if the names of these On*Xxxx* methods are different, the syntax is similar, so you can invoke these methods through delegates. This technique enables you to move the common code into a separate module and reuse it for all the properties in all your types:

```
Public Module EventHelper
   ' Delegates declaration
   Public Delegate Sub OnPropertyChangingEventHandler(Of T) _
      (ByVal e As PropertyChangingEventArgs(Of T))
   Public Delegate Sub OnPropertyChangedEventHandler(ByVal e As EventArgs)

   Public Sub AssignProperty(Of T)(ByRef oldValue As T, ByVal proposedValue As T, _
         ByVal onChanging As OnPropertyChangingEventHandler(Of T), _
         ByVal onChanged As OnPropertyChangedEventHandler)
      ' Nothing to do if the new value is the same as the old value.
      If Object.Equals(oldValue, proposedValue) Then Exit Sub
      ' Invoke the OnChangingXXXX method; exit if subscribers canceled the assignment.
      Dim e As New PropertyChangingEventArgs(Of T)(proposedValue)
      onChanging.DynamicInvoke(e)
      If e.Cancel Then Exit Sub
      ' Proceed with assignment, and then invoke the OnChangedXXXX method.
      oldValue = proposedValue
      onChanged.DynamicInvoke(EventArgs.Empty)
   End Sub
End Module
```

Thanks to the EventHelper module you can simplify the code in the Name and BirthDate properties significantly (changes are in bold type):

```
   Private m_Name As String

   Public Property Name() As String
      Get
```

```
      Return m_Name
   End Get
   Set(ByVal value As String)
      AssignProperty(Of String)(m_Name, value, AddressOf OnNameChanging, _
         AddressOf OnNameChanged)
   End Set
End Property

Private m_BirthDate As Date

Public Property BirthDate() As Date
   Get
      Return m_BirthDate
   End Get
   Set(ByVal value As Date)
      AssignProperty(Of Date)(m_BirthDate, value, AddressOf OnBirthDateChanging, _
         AddressOf OnBirthDateChanged)
   End Set
End Property
```

Object Pools

An *object pool* is a collection of objects that have been created and initialized in advance and are ready for the application to use them. Object pools are quite common in programming. For example, ADO.NET maintains a pool of connection objects: when the application asks for a connection to a database and a connection in the pool that already points to the specific database is available, ADO.NET takes a connection from the pool instead of instantiating it from scratch. When the application asks to close the connection, the physical connection isn't actually closed and the connection object is simply returned to the pool. When the same or another application asks for a connection to the same database, the connection object is taken from the pool, thus saving several seconds.

The following ObjectPool generic type implements a simple object pool. You can use this pool to create a new instance of a given type using the CreateObject method. When you don't need the object any longer, you can simply return it to the pool by using the DestroyObject method so that the next time the CreateObject method is invoked no object is physically created:

```
Public Class ObjectPool(Of T As New)
   Dim pool As New List(Of T)

   ' Create an object, taking it from the pool if possible.
   Function CreateObject() As T
      If pool.Count = 0 Then
         Return New T
      Else
         ' Return the first object to the pool.
         Dim item As T = pool(0)
         pool.RemoveAt(0)
         Return item
      End If
   End Function
```

```
   ' Return an object to the pool.
   Public Sub DestroyObject(ByVal item As T)
      pool.Add(item)
   End Sub
End Class
```

The ObjectPool class is especially useful for types that require a significant amount of time to be instantiated; under such circumstances, the application can improve performance substantially by keeping these objects alive in the pool:

```
Dim pool As New ObjectPool(Of Employee)
' These two elements are created when the method is invoked.
Dim e1 As Employee = pool.CreateObject()
Dim e2 As Employee = pool.CreateObject()
' Return one object to the pool, and then set its reference to Nothing.
pool.DestroyObject(e1)
e1 = Nothing
' Now the pool contains one element; thus, the next statement takes it from there.
Dim e3 As Employee = pool.CreateObject()
…
```

As I already explained, no form of generic constraint enables you to specify that a type must have a constructor with a given signature; thus, you can't pass arguments when instantiating a type that appears as a generic parameter. This issue severely limits the usefulness of the ObjectPool class.

The simplest way to work around this limitation and make the ObjectPool type more versatile is to define an interface that all poolable objects must implement:

```
Public Interface IPoolable
   Sub Initialize(ByVal ParamArray propertyValues() As Object)
   Function IsEqual(ByVal ParamArray propertyValues() As Object) As Boolean
End Interface
```

For each type you should define a minimum set of properties that can distinguish individual instances of that type. For example, two Employee objects should be considered as equal when their Name and BirthDate properties have the same values; therefore, the Employee class might implement the IPoolable interface as follows:

```
Public Class Employee
   Implements IPoolable

   Public Sub Initialize(ByVal ParamArray propertyValues() As Object) _
         Implements IPoolable.Initialize
      Me.Name = CStr(propertyValues(0))
      Me.BirthDate = CDate(propertyValues(1))
   End Sub

   Public Function IsEqual(ByVal ParamArray propertyValues() As Object) As Boolean _
         Implements IPoolable.IsEqual
      Return Me.Name = CStr(propertyValues(0)) AndAlso _
         Me.BirthDate = CDate(propertyValues(1))
   End Function
```

```
   ' (Implementation of Name and BirthDate properties is omitted.…)
   …
End Class
```

You can now improve the ObjectPool class to take advantage of the IPoolable interface and reuse an object in the pool only if its most important properties are equal to those of the object requested by the client:

```
Public Class ObjectPoolEx(Of T As {New, IPoolable})
   Dim pool As New List(Of T)

   ' Create an object, taking it from the pool if possible.
   Function CreateObject(ByVal ParamArray propertyValues() As Object) As T
      For i As Integer = 0 To pool.Count - 1
         Dim item As T = pool(i)
         If item.IsEqual(propertyValues) Then
            ' We've found an object with the required properties.
            pool.RemoveAt(i)
            Return item
         End If
      Next
      ' Create and return a brand-new object.
      Dim obj As New T
      obj.Initialize(propertyValues)
      Return obj
   End Function

   ' Return an object to the pool.
   Public Sub DestroyObject(ByVal item As T)
      pool.Add(item)
   End Sub
End Class
```

The code that uses the ObjectPoolEx class to create a pool of Employee objects must provide an initial value for the Name and BirthDate properties:

```
Dim pool As New ObjectPoolEx(Of Employee)
' These two elements are created when the method is invoked.
Dim e1 As Employee = pool.CreateObject("Joe", #1/1/1961#)
Dim e2 As Employee = pool.CreateObject("Ann", #2/2/1962#)
' Return them to the pool and set their references to Nothing.
pool.DestroyObject(e1)
e1 = Nothing
pool.DestroyObject(e2)
e2 = Nothing
' This object can't be taken from the pool, because its
' properties don't match any of the objects in the pool.
Dim e3 As Employee = pool.CreateObject("Joe", #3/3/1963#)
' This object matches exactly one object in the pool; thus, no new instance is created.
Dim e4 As Employee = pool.CreateObject("Ann", #2/2/1962#)
```

Once again, keep in mind that object pools are convenient only if the time you spend to instantiate an object is relevant; in all other cases, using an object pool is likely to degrade your performance without buying you any other benefit.