4

Generic Classes

Many developers will view themselves primarily as consumers of generics. However, as you get more comfortable with generics, you're likely to find yourself introducing your own generic classes and frameworks. Before you can make that leap, though, you'll need to get comfortable with all the syntactic mutations that come along with creating your own generic classes. The goal of this chapter, then, is to dig into all of the details associated with building generic classes, explaining how generics extend the existing rules for defining and consuming classes. Fortunately, as you move through this chapter, you'll notice that the syntax rules for defining generic classes follow many of the same patterns you've already grown accustomed to with non-generic types. So, although there are certainly plenty of new generic concepts you'll need to absorb, you're likely to find it quite easy to make the transition to writing your own generic types.

Parameterizing Types

In a very general sense, a generic class is really just a class that accepts parameters. As such, a generic class really ends up representing more of an abstract blueprint for a type that will, ultimately, be used in the construction of one or more specific types at run-time. This is one area where, I believe, the C++ term *templates* actually provides developers with a better conceptual model. This term conjures up a clearer metaphor for how the type parameters of a generic class serve as placeholders that get replaced by actual data types when a generic class is constructed. Of course, as you might expect, this same term also brings with it some conceptual inaccuracies that don't precisely match generics.

The idea of parameterizing your classes shouldn't seem all that foreign. In reality, the mindset behind parameterizing a class is not all that different than the rationale you would use for parameterizing a method in one of your existing classes. The goals in both scenarios are conceptually very similar. For example, suppose you had the following method in one of your classes that was used to locate all retired employees that had an age that was greater than or equal to the passed-in parameter (`minAge`):

```
[VB code]
Public Function LookupRetiredEmployees(ByVal minAge As Integer) As IList
    Dim retVal As New ArrayList
    For Each emp As Employee In masterEmployeeCollection
        If ((emp.Age >= minAge) And (emp.Status = EmpStatus.Retired)) Then
            retVal.Add(emp)
        End If
    Next
    Return retVal
End Function
```

```
[C# code]
public IList LookupRetiredEmployees(int minAge) {
    IList retVal = new ArrayList();
    foreach (Employee emp in masterEmployeeCollection) {
        if ((emp.Age >= minAge) && (emp.Status == EmpStatus.Retired))
            retVal.Add(emp);
        }
        return retVal;
    }
}
```

Now, at some point, you happen to identify a handful of additional methods that are providing similar functionality. Each of these methods only varies based on the status (Retired, Active, and so on) of the employees being processed. This represents an obvious opportunity to refactor through parameterization. By adding status as a parameter to this method, you can make it much more versatile and eliminate the need for all the separate implementations. This is something you've likely done. It's a simple, common flavor of refactoring that happens every day.

So, with this example in mind, you can imagine applying this same mentality to your classes. Classes, like methods, can now be viewed as being further generalized through the use of type parameters. To better grasp this concept, let's go ahead and build a non-generic class that will be your candidate for further generalization:

```
[VB code]
Public Class CustomerStack
    Private _items() As Customer
    Private _count As Integer

    Public Sub Push(item as Customer)
        ...
    End Sub

    Public Function Pop() As Customer
        ...
    End Function
End Class
```

```
[C# code]
public class CustomerStack {
    private Customer[] _items;
    private int _count;

    public void Push(Customer item) {...}
    public Customer Pop() {...}
}
```

This is the classic implementation of a type-safe stack that has been created to contain collections of `Customers`. There's nothing spectacular about it. But, as should be apparent by now, this class is the perfect candidate to be refactored with generics. To make your stack generic, you simply need to add a type parameter (`T` in this example) to your type and replace all of your references to the `Customer` with the name of your generic type parameter. The result would appear as follows:

```
[VB code]
Public Class Stack(Of T)
    Private _items() As T
    Private _count As Integer

    Public Sub Push(item As T)
        ...
    End Sub

    Public Function Pop() As T
        ...
    End Function
End Class
```

```
[C# code]
public class Stack<T> {
    private T[] _items;
    private int _count;

    public void Push(T item) {...}
    public T Pop() {...}
}
```

Pretty simple. It's really not all that different than adding a parameter to a method. It's as if generics have just allowed you to widen the scope of what your classes can parameterize.

Type Parameters

By now, you should be comfortable with the idea of type parameters and how they serve as a type placeholder for the type arguments that will be supplied when your generic class is constructed. Now let's look at what, precisely, can appear in a type parameter list for a generic class.

First, let's start with the names that can be assigned to type parameters. The rules for naming a type parameter are similar to the rules used when defining any identifier. That said, there are guidelines that you should follow in the naming of your type parameters to improve the readability and maintainability of your generic class. These guidelines, and others, are discussed in Chapter 10, "Generics Guidelines."

A generic class may also accept multiple type parameters. These parameters are provided as a delimited list of identifiers:

```
[VB code]
Public Class Stack(Of T, U, V)
```

```
[C# code]
public class Stack<T, U, V>
```

As you might suspect, each type parameter name must be unique within the parameter list as well as within the scope of the class. You cannot, for example, have a type parameter `T` along with a field that is also named `T`. You are also prevented from having a type parameter and the class that accepts that parameter share the same name. Fortunately, the names you're likely to use for type parameters and classes will rarely cause collisions.

In terms of scope, a type parameter can only be referenced within the scope of the generic class that declared it. So, if you have a child generic class `B` that descends from generic class `A`, class `B` will not be able to reference any type parameters that were declared as part of class `A`.

The list of type parameters may also contain constraints that are used to further qualify what type arguments can be supplied of a given type parameter. Chapter 7, "Generic Constraints," will look into the relevance and application of constraints in more detail.

Overloaded Types

The .NET implementation of generics allows programmers to create overloaded types. This means that types, like methods, can be overloaded based on their type parameter signature. Consider the declarations of the following types:

```
[VB code]
Public Class MyType
   ...
End Class

Public Class MyType(Of T)
    ...
End Class

Public Class MyType(Of T, U)
    ...
End Class
```

```
[C# Code]
public class MyType {
}

public class MyType<T> {
    ...
}

public class MyType<T, U> {
    ...
}
```

Three types are declared here and they all have the same name and different type parameter lists. At first glance, this may seem invalid. However, if you look at it from an overloading perspective, you can see how the compiler would treat each of these three types as being unique. This can introduce some level of confusion for clients, and this is certainly something you'll want to factor in as you consider building your own generic types. That said, this is still a very powerful concept that, when leveraged correctly, can enrich the power of your generic types.

Static Constructors

All classes support the idea of a static (shared) constructor. As you might expect, a static constructor is a constructor that can be called without requiring clients to create an instance of a given class. These constructors provide a convenient mechanism for initializing classes that leverage static types.

Now, when it comes to generics, you have to also consider the accessibility of your class's type parameters within the scope of your static constructor. As it turns out, static constructors are granted full access to any type parameters that are associated with your generic classes. Here's an example of a static constructor in action:

```
[VB code]
Imports System.Collections.Generic

Class MySampleClass(Of T)
    Private Shared _values As List(Of T)

    Shared Sub New()
        If (GetType(T).IsAbstract = False) Then
            Throw New Exception("T must not be abstract")
        Else
            _values = New List(Of T)()
        End If
    End Sub
End Class
```

```
[C# code]
using System.Collections.Generic;

public class MySampleClass<T> {
    private static List<T> _values;

    static MySampleClass() {
        if (typeof(T).IsAbstract == false)
            throw new Exception("T must not be abstract");
        else
            _values = new List<T>();
    }
}
```

This example creates a class that accepts a single type parameter, `T`. The class has a data member that is used to hold a static collection of items of type `T`. However, you want to be sure, as part of initialization, that `T` is never abstract. In order to enforce this constraint, this example includes a static constructor that examines the type information about `T` and throws an exception if the type of `T` is abstract. If it's not abstract, the constructor proceeds with the initialization of its static collection.

This is just one application of static constructors and generic types. You should be able to see, from this example, how static constructors can be used as a common mechanism for initializing any generic class that has static data members.

Inheritance

Generic concepts, of course, are not constrained to a single class declaration. The type parameters that are supplied with a generic class declaration can also be applied to all the objects that participate in an object hierarchy. Here's a simple example that creates a generic base class and subclasses it with another generic class:

[VB code]

```
Public Class MyBaseClass(Of U)
    Private _parentData As U

    Public Sub New()
        ...
    End Sub

    Public Sub New(ByVal val As U)
        Me._parentData = val
    End Sub
End Class

Public Class MySubClass(Of T, U)
    Inherits MyBaseClass(Of U)
    Private _myData As T

    Public Sub New()
        ...
    End Sub

    Public Sub New(ByVal val1 As T, ByVal val2 As U)
        MyBase.New(val2)
        Me._myData = val1
    End Sub
End Class
```

[C# code]

```
public class MyBaseClass<U> {
    private U _parentData;

    public MyBaseClass() {}

    public MyBaseClass(U val) {
        this._parentData = val;
    }
}

public class MySubClass<T, U> : MyBaseClass<U> {
    private T _myData;

    public MySubClass() {}

    public MySubClass(T val1, U val2) : base(val2) {
        this._myData = val1;
    }
}
```

Notice here that you have `MyBaseClass`, which accepts a single type parameter. Then, you subclass `MyBaseClass` with `MySubClass`, which accepts two type parameters. The key bit of syntax to notice here is that one of the type parameters used in the declaration of `MySubClass` was also referenced in the declaration of the parent class, `MyBaseClass`.

Although this example simply passed the type parameters from the subclass to the base class, you can also use type arguments when inheriting from another class or implementing a generic interface. In this case, your generic class declarations could appear as follows:

```
[VB code]
Public Class MyBaseClass(Of U)
    ...
End Class

Public Class MySubClass(Of T)
    Inherits MyBaseClass(Of Integer)
    ...
End Class
```

```
[C# code]
public class MyBaseClass<U> {
}

public class MySubClass<T> : MyBaseClass<int> {
}
```

The subclass has been altered here and now only accepts a single type parameter. And, in place of the type parameter `U` that was being passed to `MyBaseClass`, you now pass the type argument `Integer`. After looking at these two examples, it should be clear that your classes can subclass another class using both open and constructed types (and some variations in between). It's probably obvious at this stage, but I should point out that your generic classes can also subclass non-generic, closed base classes as well.

Now that you know what works, let's take a quick look at some of the combinations of inheritance patterns that will not work. Suppose you have a closed type that inherits from a generic class:

```
[VB code]
Public Class MyBaseClass(Of T, U)
End Class

Public Class MySubClass1
    Inherits MyBaseClass(Of T, U)
End Class

Public Class MySubClass2
    Inherits MyBaseClass(Of Int32, String)
End Class
```

```
[C# code]
public class MyBaseClass<T, U> { }

public class MySubClass1 : MyBaseClass<T, U> { }

public class MySubClass2 : MyBaseClass<int, string> { }
```

In this example, you have two closed types that inherit from a generic class. `MySubClass1` attempts to inherit using an open type and `MySubClass2` inherits as a constructed type. When you attempt to compile this code, you're going to notice that `MySubClass1` will generate an error. The compiler has no point of reference that allows it to resolve the types of `T` and `U`. As a constructed type, `MySubClass1` doesn't accept any type parameters and, therefore, has no parameters that can be used because it inherits from `MyBaseClass<T, U>`. `MySubClass2` doesn't accept type parameters either, but it still compiles because it uses type arguments and forms a constructed type as part of its inheritance declaration.

There's one more inheritance scenario that's worth discussing here. Let's look at an example where you use a constructed type as a type argument in the declaration of your inherited class:

```
[VB code]
Imports System.Collections.Generic

Public Class MyBaseClass(Of T, U)
End Class

Public Class MySubClass1(Of T)
    Inherits MyBaseClass(Of List(Of T), T)
End Class

Public Class MySubClass2(Of T)
    Inherits MyBaseClass(Of List(Of T), Stack(Of T))
End Class
```

```
[C# code]
using System.Collections.Generic;

public class MyBaseClass<T, U> { }

public class MySubClass1<T> : MyBaseClass<List<T>, T> { }

public class MySubClass2<T> : MyBaseClass<List<T>, Stack<T>> { }
```

If you look at this example closely, you'll discover that it's really just a variation on one of the earlier examples of inheritance. The key difference here is that a mixture of constructed types and type parameters are used where the constructed types end up leveraging the type parameter from the subclass. The goal here is just to get you comfortable with the possibilities and to get you to view a constructed type like any other type argument you might pass when inheriting from a generic class.

Finally, it's worth noting that a generic class cannot use one of its type parameters as its inherited type. It must descend from an existing closed or open type. For example, the following would not be legal:

```
[VB code]
Public Class MyType(Of T)
   Inherits Of T
   ...
End Class
```

```
[C# code]
public class MyType<T> : T {
    ...
}
```

At this stage, after looking at all these examples, you should have a much better idea for how the mechanics of generic inheritance work. The rules that govern inheritance are fairly logical and don't really fall outside what you might expect. The main idea to take away here is that you can use both type parameters and type arguments as parameters when subclassing a generic class.

Protected Members

Any discussion of inheritance would be incomplete without also examining the accessibility of protected members. First, I should make it clear that all the rules that govern access to protected members are unchanged for closed types. Things get a bit more interesting when you look at protected members that appear within a generic class. These members are accessible too. In fact, they are accessible in some ways you might not expect.

Here's an example that declares a base class with a protected, generic field that will then be referenced in a descendant class:

[VB code]

```
Public Class MyBaseClass(Of T)
    Protected _var1 As T

    Public Sub New(ByVal var1 As T)
        Me._var1 = var1
    End Sub
End Class

Public Class MySubClass(Of T, U)
    Inherits MyBaseClass(Of T)
    Private _var2 As U

    Public Sub New(ByVal var1 As T, ByVal var2 As U)
        MyBase.New(var1)
        Me._var2 = var2
    End Sub

    Private Sub Foo1()
        Dim localVar as T = Me._var1
    End Sub

    Private Sub Foo2()
        Dim sub1 As New MySubClass(Of Integer, String)(1, "12")
        Dim val1 As Integer = sub1._var1
        Dim sub2 As New MySubClass(Of Double, Double)(1.0, 5.8)
        Dim val2 As Double = sub2._var1
        End Sub
End Class
```

[C# code]

```
using System;

public class MyBaseClass<T> {
    protected T _var1;

    public MyBaseClass(T var1) {
```

```
        this._var1 = _var1;
    }
}

public class MySubClass<T, U> : MyBaseClass<T> {
    private U _var2;

    public MySubClass(T var1, U var2) : base(var1) {
        this._var2 = var2;
    }

    private void Foo1() {
        T localVar = this._var1;
    }

    private void Foo2() {
        MySubClass<int, String> sub1 = new MySubClass<int, String>(1, "12");
        int val1 = sub1._var1;
        MySubClass<Double, Double> sub2 = new MySubClass<Double, Double>(1.0, 5.8);
        Double val2 = sub2._var1;
    }
}
```

This example actually ends up illustrating two key points. It includes a protected field in the base class, `var1`, that is then accessed by the subclass. The method `Foo1()` accesses `var1` successfully, much like it would any other inherited, protected data member. The method `Foo2()` is the more interesting example. It constructs two separate instances of `MySubClass` and, because it's in the scope of the descendant class, it is able to access the protected member `var1` via these constructed types.

Fields

The fields of a generic class follow all the same syntax rules that are applied to non-generic fields. The primary incremental change here is the ability to reference type parameters in the declaration of your class's fields. Here's an example where this is applied:

```
[VB code]
Public Class MyType(Of T, U)
    Private _myFirstDataMember As T
    Private _mySecondDataMember As U
    Public Sub New(ByVal val1 As T, ByVal val2 As U)
        Me._myFirstDataMember = val1
        Me._mySecondDataMember = val2
    End Sub

    Public Function GetFirstDataMember() As T
        Return Me._myFirstDataMember
    End Function

    Public Function GetSecondDataMember() As U
        Return Me._mySecondDataMember
```

```
    End Function
End Class

Public Class MyApp
    Shared Sub Main()
        Dim testType As New MyType(Of String, String)("val1", "Val2")
        Console.WriteLine(testType.GetFirstDataMember())
        Console.WriteLine(testType.GetSecondDataMember())
    End Sub
End Class
```

```
[C# code]
using System;

class MyType<T, U> {
    private T _myFirstDataMember;
    private U _mySecondDataMember;

    public MyType(T val1, U val2) {
        this._myFirstDataMember = val1;
        this._mySecondDataMember = val2;
    }

    public T GetFirstDataMember() {
        return this._myFirstDataMember;
    }

    public U GetSecondDataMember() {
        return this._mySecondDataMember;
    }
}

class MyApp {
    static void main(String[] args) {
        MyType<string, string> testType =
                        new MyType<string, string>("Val1", "Val2");
        Console.WriteLine(testType.GetFirstDataMember());
        Console.WriteLine(testType.GetSecondDataMember());
    }
}
```

As you can see, the generic class is able to create a pair of fields that reference the type parameters that are part of the class declaration. In fact, the type of any field declared in your generic class can reference any type parameter that is passed into your class. You'll also notice that the constructor initializes these fields using the same syntax you would use for any non-generic field.

Static Fields

Although type parameters can be used in the place of most types in a class, they cannot be applied to any static field. In fact, the behavior of static fields does not change within a generic class. Consider the following implementation of an object cache:

[VB code]

```
Imports System.Collections.Generic

Public Class MyCache(Of K, V)
    Private Shared _objectCache As New Dictionary(Of K, V)

    Public Sub New()
    End Sub

    Public Function FindValueInDB(ByVal key As K) As V
        'findValue (not shown) would lookup the key in
        'a repository, add it to our cache, and return the value
    End Function

    Public Function LookupValue(ByVal key As K) As V
        Dim retVal As V
        If (_objectCache.ContainsKey(key) = True) Then
            _objectCache.TryGetValue(key, retVal)
        Else
            retVal = FindValueInDB(key)
        End If
        Return retVal
    End Function
End Class

Public Class MyApp
    Public Shared Sub Main()
        Dim cache1 As New MyCache(Of String, String)()
        Dim val1 As String = cache1.LookupValue("key1")

        Dim cache2 As New MyCache(Of String, String)()
        Dim val2 As Integer = cache2.LookupValue("key1")
    End Sub
End Class
```

[C# code]

```
using System.Collections.Generic;

class MyCache<K, V> {
    private static Dictionary<K, V> _objectCache;

    public MyCache() {
        MyCache<K, V>._objectCache = new Dictionary<K, V>();
    }

    private V findValueInDB(K key) {
        // findValue (not shown) would lookup the key in
        // a repository, add it to our cache, and return the value
        return default(V);
    }

    public V lookupValue(K key) {
        V retVal;
        if (_objectCache.ContainsKey(key) == true) {
            _objectCache.TryGetValue(key, out retVal);
        } else {
```

```
            // findValue (not shown) would lookup the key in
            // a repository, add it to our cache, and return the value
            retVal = findValueInDB(key);
        }
        return retVal;
    }
}

class MyApp {
    public static void main(String[] args) {
        MyCache<string, string> cache1 = new MyCache<string, string>();
        string val1 = cache1.lookupValue("key1");

        MyCache<string, int> cache2 = new MyCache<string, int>();
        int val2 = cache2.lookupValue("key1");
    }
}
```

This example defines a generic class, `MyCache`, that employs a static field to hold a set of keys and their corresponding values. It represents a simple wrapper for the generic `Dictionary` class, adding a single `lookupValue` method. Each time this method is called, it will determine if the provided key already exists in the cache. If the key is found, the corresponding value will be returned. If the key is not found, the code class will look it up in a database, place it in the cache, and return it to the client.

The goal here was to construct a simple cache that could make this added lookup capability available to a wide variety for data types. You could, for example, leverage this class to hold a cache strings and turn around and reuse it, as I did here, to manage a *separate* cache of integers. And, in that regard, it delivers what I had intended. The problem comes in when two constructed types use the same type arguments.

In these instances, your constructed types actually end up sharing more than their implementation — they also share their static fields. So, although it seemed as though your two constructed types represented unique instances, they did not. They both shared a single instance from the static `objectCache` field.

You can only imagine what this does to your cache. If you declare two constructed types with matching type arguments and you want them treated entirely separately, it won't be possible. The static cache being maintained by the class will end up being shared by these two instances.

In general, this aspect of static fields shouldn't be viewed as a limitation. It's mostly significant and worth discussing because the behavior may not match what you're expecting. It could also be the source of a few difficult-to-locate bugs. If nothing else, this is at least something you'll want to keep in mind whenever you opt to include static data in one of your generic classes.

Constructed Fields

It should go without saying that constructed generic types can also be used throughout the implementation of your generic classes. Specifically, you are allowed to have fields that are open or closed constructed types. Here's a quick look at a simple example:

```
[VB code]
Imports System.Collections.Generic

Public Class MyType(Of T)
```

```
    Private _myList As New List(Of T)
    Private _myStrings As New List(Of String)

    Public Function GetItem(ByVal param1 As List(Of Integer)) As T
        Dim localVar As New List(Of T)
        Return _myList(0)
    End Function
End Class
```

```
[C# code]
using System.Collections.Generic;

class MyType<T> {
    private List<T> _myList;
    private List<string> _myStrings;

    public T getItem(List<int> param1) {
        List<T> localVar;
        return _myList[0];
    }
}
```

You'll notice the class is littered with open and closed constructed types. After you're familiar with generics, these constructed types will start to look like any other data type you might reference in your class. The syntax variations for generics seem to just fade into the background.

Methods

Methods represent your primary point of contact with clients. As such, their flexibility, expressiveness, and general type safety should be of a great deal of importance to you. You want to make your client's life as simple as possible and, because you may likely be both the producer and the consumer of these methods, you're also likely to be especially motivated to build a good interface.

With generics, you actually have the opportunity to build a set of interfaces that are likely to last more than a few days. If you think about it, generics allow you to be 100% vague about types that are supported by your interfaces. It's as if every parameter and every return type is of the type `object`. All that, and you still get a type-safe interface.

Okay, that may be a bit extreme. But, if you consider that your type parameters can essentially appear anywhere within the signature of a method, it doesn't seem like that big of a stretch. At the same time, while you feel this extra degree of freedom in defining your interface, the consumers of your interface get to work with specific types without dealing with casting or conversion.

> **For this chapter, my focus is exclusively on generic classes. And, for that reason, the discussion of methods is intentionally constrained to how type parameters influence the methods of your class. Chapter 5, "Generic Methods," looks at generic methods that can exist entirely outside the scope of a generic class and have their own set of dynamics.**

Let's take a quick look at a sample class that illustrates a few variations on how you might go about using type parameters in the interface of your generic class:

```
[VB code]
Class MyType(Of V)
End Class

MustInherit Class MyShape(Of T, U)
    Public Sub New()
    End Sub

    MustOverride Sub Draw()
    MustOverride Function GetIndexes() As T()

    Overridable Function
            AddValue(ByVal index As T, ByVal value As MyType(Of U)) As Boolean
        ...
    End Function

    Overridable Function GetValue(ByVal index As T) As MyType(Of U)
        ...
    End Function
End Class
```

```
[C# code]
class MyType<V> {}

abstract class MyShape<T, U> {
    public MyShape() {}

    public abstract void draw();
    public abstract T[] GetIndexes();

    public virtual bool AddValue(T index, MyType<U> value) {

        ...
    }

    public virtual MyType<U> GetValue(T index) {
        ...
    }
}
```

In this class you define a set of methods, both abstract and virtual, that leverage the type parameters used in the declaration of your generic class. For both the abstract and virtual methods, you are free to include type parameters and open types wherever you would place traditional types. As you can see, very few limitations are imposed on the methods of your class. And, fortunately, most of the concepts you've become accustomed to with non-generic methods will still apply to your methods.

Overloading Methods

Whenever you define a method — especially an overloaded method — you need to understand how the compiler will evaluate the signature of that method. Specifically, you need to have a firm grasp on how

the compiler evaluates the uniqueness of a given generic method. For example, consider how the compiler will evaluate the following two methods:

```
 [VB code]
Class MyType(Of T)
    Public Function Foo() As Boolean
    End Function

    Public Function Foo() As T
    End Function
End Class
```

```
[C# code]
class MyType<T> {
    public bool Foo() { ... }
    public T Foo() { ... }
}
```

While these two methods would seem to have different signatures, the compiler has no means by which it can uniquely identify these two methods because they only differ by their return types — one of which is a type parameter. This is not especially surprising because non-generic methods also fail when they only differ by their return types. Here's a less obvious example:

```
[VB code]
Class MyType(Of T)
    Public Function Foo(ByVal myString As String) As Boolean
    End Function

    Public Function Foo(ByVal myValue As T) As Boolean
    End Function
End Class
```

```
[C# code]
class MyType<T> {
    public bool Foo(String myString) {  }
    public bool Foo(T myValue) {  }
}
```

On the surface, it would seem as though the compiler would not be able to distinguish between these two methods. However, this is one where C# and VB vary in their implementations. The C# generics specification only requires a method's signature to be unique prior to the class's instantiation. So, given these constraints, the C# compiler is able to view these two methods as being unique. VB, on the other hand, isn't so kind. It continues to throw an error and complain about the uniqueness of the method's signature.

Although the C# language may support this syntax, it still creates a situation that could cause some level of confusion for clients of your method. As such, relying on this mechanism may not be a good idea — especially given the overriding generic theme of providing clear, expressive interfaces for your types.

The name you assign to your type parameters can also create some confusion when you are trying to evaluate the signature of a method. Typically, you look at the name of a type to determine if two types

match. For example, in the next snippet of code you have two `Integer` parameters in the signature of two separate methods:

```
[VB code]
Class MyType(Of T)
    Public Sub Foo(val1 As Integer, val2 As Integer)
    End Sub

    Public Sub Foo(val1 As Integer, val2 As Integer)
    End Sub
End Class
```

```
[C# code]
class MyType<T> {
    public void Foo(int val1, int val2) { ... };
    public void Foo(int val1, int val2) { ... };
}
```

To most developers going to be, it's clear that these two methods are going to collide and throw a compile-time error. Now, let's take that same concept and use type parameters instead of native types:

```
[VB code]
Class MyType(Of T, U, V)
    Public Sub Foo(ByVal val1 As T, ByVal val2 As V)
    End Sub

    Public Sub Foo(ByVal val1 As U, ByVal val2 As V)
    End Sub
End Class
```

```
[C# code]
class MyType<T, U, V> {
    public void Foo(T val1, V val2) { ... }
    public void Foo(U val1, V val2) { ... }
}
```

These two methods, based on type parameter names alone, would appear to have unique signatures. There are certainly scenarios where, with different permutations of type arguments, these signatures would be deemed unique. At the same time, it's also true that there could be combinations of type arguments that could create collisions.

It turns out the VB and C# take different approaches to verifying the uniqueness of these signatures. VB says if there can be at least one combination of type arguments that *could* cause these methods to be duplicates, then the compiler will throw an error. This will be true even if there are no constructed types in your code that create a collision. C#, on the other hand, takes the more optimistic approach. It allows this class to compile because there are combinations of type arguments that could be valid.

Overriding Methods

Now that you have a better idea of how type parameters influence the signature of methods, you need to consider how type parameters are applied when you override methods in a generic class. First, it's important to point out that, if your generic class descends from a closed type, you can still override the parent methods just as you would in any non-generic class.

Where this gets more interesting is when your base class is generic class (open or constructed type). For this scenario, you can still override methods from your generic base class. However, there are some nuances you must keep in mind. Let's start by looking at the common, simple case:

```
[VB code]
Public Class MyBaseClass(Of T)
        Overridable Sub Foo(ByVal val As T)
            Console.WriteLine("In BaseClass")
        End Sub
    End Class

    Public Class MySubClass1(Of T, U)
        Inherits MyBaseClass(Of T)

        Overrides Sub Foo(ByVal val As T)
            Console.WriteLine("In SubClass")
        End Sub
    End Class

    Public Class MySubClass2(Of T, U)
        Inherits MyBaseClass(Of Integer)

        Overrides Sub Foo(ByVal val As T)
            Console.WriteLine("In SubClass")
        End Sub
End Class
```

```
[C# code]
public class MyBaseClass<T> {
    public virtual void Foo(T val) { }
}

public class MySubClass1<T, U> : MyBaseClass<T> {
    public override void Foo(T val) { }
}

public class MySubClass2<T, U> : MyBaseClass<int> {
    public override void Foo(T val) { }
}
```

This example declares a generic base class (`MyBaseClass`) that includes one overridable method, `Foo`. It then implements two generic subclasses that both override the `Foo` method. On the surface, there doesn't appear to be any issues. Both of these classes provide identical signatures for the method. So, why does `MySubClass2` fail to compile? Well, if you look more closely, you'll notice that `MySubClass2` uses a constructed type in its inheritance declaration. Meanwhile, `MySubClass1` uses an open type for its declaration. This one point of difference is crucial. With `MySubClass1`, the type parameter `T` used inheritance declaration and the overridden method can be guaranteed to match. That doesn't hold true for `MySubClass2`. The type of `T` can, and likely will, differ from the integer type, which is what is provided to the parent's `T` type parameter. As you might expect, the compiler is going to detect this and throw an error during the compilation of `MySubClass2`.

The theme here is that the type parameters and type arguments supplied to an inherited generic class play a significant role in defining what's legal when overriding a method. As type parameters are referenced in overridable methods of the parent class, those type parameters must be resolvable to the same type of the overriding base class. Here's one more example to solidify this point:

```
[VB code]
Public Class MyBaseClass(Of T, U)
    Overridable Sub Foo(ByVal val1 As T, ByVal val2 As U)
        Console.WriteLine("In BaseClass")
    End Sub
End Class

Public Class MySubClass1(Of T, U, V)
    Inherits MyBaseClass(Of T, V)

    Overrides Sub Foo(ByVal val1 As T, ByVal val2 As U)
        Console.WriteLine("In SubClass")
    End Sub
End Class
```

```
[C# code]
public class MyBaseClass<T, U> {
    public virtual void Foo(T val1, U val2) { }
}

public class MySubClass1<T, U, V> : MyBaseClass<T, V> {
    public override void Foo(T val2, U val2) { }
}
```

Once again, the signatures of the methods seem to match. And, once again, the compiler isn't happy. The mismatch here is that the subclass supplies `T` and `V` as type arguments to the parent's corresponding `T` and `U` type parameters. This means that the `U` type parameter in the base class actually maps to the `V` type parameter in the subclass. So, when you reference the `T` and `U` parameters in the overriding method, the `U` parameter actually represents `V` and causes a compile error.

The rules for overriding in generic classes aren't much more involved than that. You won't find yourself getting tripped up by this too often. And, when you do, the compiler does a reasonable job detecting and reporting errors in this scenario.

Arrays of Type Parameters

The methods within your generic class may also include arrays of type parameters. This gives you the ability to pass type-safe arrays to your methods. This syntax is as follows:

```
[VB code]
Class MyType(Of T)
    Shared Sub Foo(ByVal params As T())
    End Sub
End Class

Public Class MyTest
    Public Shared Sub Test()
```

```
        MyType(Of Integer).Foo(New Integer() {123, 321})
        MyType(Of String).Foo(New String() {"TEST1", "TEST2", "TEST3"})
    End Sub
End Class
```

```
[C# code]
public class MyType<T> {
    public static void Foo(T[] parms) { }
}
public class MyTest {
    public static void Test() {
        MyType<int>.Foo(new int[] { 123, 321 });
        MyType<string>.Foo(new string[] { "TEST1", "TEST2", "TEST3" });
    }
}
```

This works exactly as you might expect. In fact, the syntax matches that of any other array you may pass to a method.

Operator Overloading

For some reason, operator overloading seems particularly interesting when it comes to generics. Here you have this new generic class and now you're allowed to define operations on that class without any awareness of the types it will be managing. Not sure why, but that's just plain cool to me. It's as if you can define all these semantics of your generic types at an all new level of abstraction.

The other thing to be excited about here is the enhanced operator overloading support in Visual Studio 2005. Specifically, Visual Basic finally has *real* support for operator overloading that allows Visual Basic to define operators that are more in line with the traditional model that has been historically provided as part of other languages.

All that said, the important part here is to understand the syntax rules that govern the definition and invocation of overloaded operators within generic classes.

```
[VB code]
Public Class MyType(Of T)
    Public Shared Operator +(ByVal op As MyType(Of T)) As MyType(Of T)
        Console.WriteLine("In unary ++ operator")
        Return New MyType(Of T)()
    End Sub
End Class
```

```
[C# code]
using System;

public class MyType<T> {
    public static MyType<T> operator ++(MyType<T> op) {
        Console.WriteLine("In ++ operator");
        return new MyType<T>();
    }
}
```

This example implements the + unary operator. You can see here that this example simply substitutes the generic open type, `MyType`, in the appropriate locations in the signature of the operator method. This is a requirement for your overloaded generic operators. In fact, all unary operations must take one parameter of the instance type.

Any time there's a discussion of operator overloading, it must also be accompanied by discussion of type conversion. It makes sense. After all, operator overloading is what enables you to provide specific conversion operators that determine how one type can be converted to another. For example, if you cast `MyType1` to `MyType2`, an overloaded operator could be used to define the behavior of this conversion (assuming you want this to be a valid conversion).

In the world of generics, you can't really define conversion from one concrete type to another. There's nothing concrete about your types at all. That's the whole point. The syntax of generics, however, does provide you with the mechanisms you'll need to express these conversions in a generic fashion. Here's a quick example of an overloaded operator that provides type conversion:

```
[VB code]
Public Class MyType(Of T)
    Public Shared Widening Operator CType(ByVal source As MyType(Of T)) _
        As MyType(Of String)

        Console.WriteLine("In unary string conversion operator")
        Return New MyType(Of T)()
    End Sub
End Class
```

```
[C# code]
public class MyType<T> {
    public static implicit operator MyType<String>(MyType<T> source) {
        Console.WriteLine("In unary string conversion operator");
        return new MyType<String>();
    }
}
```

This example provides an operator that will convert instances of `MyType` to constructed type of `MyType(Of String)`. This can be fairly handy and provides an excellent alternative to providing operator overloads for every possible source type.

There are a few gotchas you'll want to plant in your memory when providing conversion operators. First, you cannot perform conversions if your source and target conversion types are in the same object hierarchy. Also, it's important to note that your conversion operators may end up overloading a conversion operator that is already defined. If this turns out to be the case, your overloaded conversion operator will never end up getting called.

Nested Classes

Generic classes do not fundamentally change the nature of nested classes. Of course, by definition, any class that's nested inside a generic class is also deemed "generic" in that the overall constructed type cannot be created without someone providing type parameters.

That said, generics also extend the functionality of existing nested classes, allowing them to have full access to the type parameters of their *outer* class. In addition, you also have the option of making your nested classes be generic. Following is an example where a nested class references the type parameters of its outer class:

```
[VB code]
Imports System.Collections.Generic

Public Class OuterClass(Of T, U)
    Public var1 As New Dictionary(Of T, U)

    Public Class InnerClass
        Private var1 As T
        Private var2 As New List(Of U)
    End Class
End Class
```

```
[C# code]
using System.Collections.Generic;

public class OuterClass<T, U> {
    public Dictionary<T, U> var1;

    public class InnerClass {
        private T var1;
        private List<U> var2;
    }
}
```

As you can see, the declaration of your inner class matches that of any other nested class. However, within your inner class, you'll notice that the example declares a few data members that use the type parameters from your outer class. In fact, the example declares a number of different constructed types from the pool of type parameters that were supplied to your generic class.

This gets a little more interesting when you make your inner class accept its own type parameters. Here's an example that does just that:

```
[VB code]
Imports System.Collections.Generic

Public Class OuterClass(Of T, U, V)

    Public Class InnerClass(Of T, V)
        Private var1 As New Dictionary(Of T, V)
        Private var2 As U
    End Class
End Class
```

```
[C# code]
using System.Collections.Generic;

public class OuterClass<T, U, V> {
    public class InnerClass<T, V> {
        private Dictionary<T, V> var1;
```

```
        private U var2;
    }
}
```

The outer class for this example accepts three type parameters: `T`, `U`, and `V`. Your inner class also accepts type parameters named `T` and `V`. So, the question is, what will happen with the `var1` data member that you've declared in your inner class? Will it share the same types that are supplied to the outer class for the `T` and `V` type parameters? Nope. Once you used these type parameter names for your inner class, you effectively lost all ability to reference the type parameters from your outer class. They are inaccessible. Meanwhile, the other data member in your inner class, `var2`, is able to successfully reference the `U` type parameter from the outer class. This is possible because `U` was not included in the type parameter list of the inner class.

This problem would mostly be chalked up to bad habits. As a rule of thumb, the creator of an inner class should never reuse the type parameter names of its outer class as part of its declaration. Otherwise, you will be forever and unnecessarily prevented from accessing the type parameters of your outer class. You don't want to live with that kind of guilt.

Consuming Generic Classes

Okay, you've had a good long look at what goes into defining a generic class. Now, it's time to explore those rules that govern constructed types. First, it should be clear by now that a constructed type shares all the freedoms as any other type and can be placed in any syntactical context that would be used for non-generic types. Once you marry a type argument to the open type, they are conceptually "merged" to form a specific concrete type. The sooner you're comfortable with that notion, the sooner you'll begin to view constructed types on equal ground with `String`s and `Integer`s (actually, given my bias, I would tend to view them as generally superior to these types on the sheer merit of their constitution).

A number of variations exist on how you might declare a constructed type. The simplest variety is what is considered a closed constructed type. It's simple because the type arguments supplied are of simple, concrete types. Open constructed types use the type parameters from their surrounding generic class as part of their declaration. These two types are likely familiar by now. However, Chapter 1, "Generics 101," provides examples if you want a more detailed explanation.

In addition to accepting all the primitive types as type arguments, a constructed type can also be created using other constructed types as type arguments. For example:

```
[VB code]
Dim myGenericType1 As New MyType(Of List(Of Integer))
Dim myGenericType2 As New MyType2(Of Dictionary(Of Integer, String), String)

[C# code]
MyType1<List<int>> myGenericType1;
MyType2<Dictionary<int, string>, string> myGenericType2;
```

This fits with the theme that a constructed type is just like any other type and, as such, can behave like any other type argument. You also have the option of using type parameters in these declarations, which would simply make your declaration an open constructed type (because its type will be determined at run-time). It's also worth noting that these types can also be passed as arrays by applying the array modifier to these arguments.

Accessibility

Whenever you look at introducing new syntax for types, you must also look at the rules that govern accessibility for these new types. Each time you declare a constructed type, the accessibility of that type's parameters must be taken into consideration. Here's a basic example that highlights how accessibility can influence a constructed type:

```
[VB code]
Public Class MyType(Of T)

End Class

Public Class AccessTest
    Private Class PrivateClass
    End Class

    Public Function GetMyType() As MyType(Of PrivateClass)
    End Function
End Class
```

```
[C# code]
public class MyType<T> {}

public class AccessTest {
    private class PrivateClass {}
    public MyType<PrivateClass> getMyType() { }
}
```

This example creates a closed class, `AccessTest`, which contains a nested class (`PrivateClass`) as well as a public method. You'll notice that this method actually returns a constructed type, `MyType`, which was constructed using `PrivateClass` as a type argument.

The problem with this implementation is that `PrivateClass`, which has private accessibility, is being used in the construction of the publicly accessible return type of the method `getMyType()`. As you might suspect, the compiler catches and throws an error when you attempt to compile this example. This same brand of error would also apply in situations where you attempt to use a protected type as a parameter to a publicly accessible constructed type.

The rule of thumb here is that the accessibility of any class's constructed types is constrained by the accessibility of the type arguments passed to that constructed type. Let's look at another example:

```
[VB code]
Public Class MyType(Of T)
End Class

Public Class AccessTest
    Public class PublicClass
    End Class
    Protected class ProtectedClass
    End Class
    Private class PrivateClass
    End Class

    Public Function Foo1() As MyType(Of ProtectedClass)
```

```
        End Function

        Protected Function Foo2 As MyType(Of ProtectedClass)
        End Function

        Protected Function Foo3 As MyType(Of PublicClass)
        End Function

        Private Function Foo4 As MyType(Of PrivateClass)
        End Function
    End Class
```

```
     [C# code]
    class MyType<T> {}
    class AccessTest {
        public class PublicClass {}
        protected class ProtectedClass {}
        private class PrivateClass {}
        public MyType<ProtectedClass> Foo1() { }
        protected MyType<ProtectedClass> Foo2() { }
        protected MyType<PublicClass> Foo3() { }
        private MyType<PrivateClass> Foo4() { }
    }
```

In this example, three classes are declared — `PublicClass`, `ProtectedClass`, and `PrivateClass` — and they are passed as type arguments to construct return value types for four different methods. The first method, `Foo1()`, passes a `ProtectedClass` to a constructed type with public accessibility, which, as you might expect, yields a compile error. This publicly accessible method cannot return a constructed type that has been constructed with a protected type. The remaining methods will all successfully compile because the accessibility of their type arguments do not "exceed" the accessibility of each method.

These scenarios seem pretty straightforward. However, there's one more variation that's worth exploring. What happens in the instances where you have multiple type arguments being passed to your constructed type, each with varying levels of accessibility? Consider the following example:

```
    [VB code]
    Public Class AType(Of T, U)
    End Class

    Public Class MyType
        Public Class PublicClass
        End Class
        Protected Class ProtectedClass
        End Class
        Protected Friend Class ProtectedFriendClass
        End Class
        Private Class PrivateClass
        End Class

        Public Function Foo1() As AType(Of PublicClass, PublicClass)
        End Function
        Protected Function Foo2() As AType(Of ProtectedClass, ProtectedFriendClass)
        End Function
        Protected Friend Function Foo3() As AType(Of PublicClass, ProtectedFriendClass)
```

```
        End Function
        Friend Function Foo4() As AType(Of PublicClass, PublicClass)
        End Function
        Private Function Foo5() As AType(Of PublicClass, PrivateClass)
        End Function

        Public Function Foo6() As AType(Of ProtectedClass, PublicClass)
        End Function
        Protected Function Foo7() As AType(Of ProtectedClass, PrivateClass)
        End Function
        Protected Friend Function Foo8() As AType(Of ProtectedClass, PublicClass)
        End Function
    End Class
```

```
    [C# code]
    class AType<T, U> {}
    class MyType {
        public class PublicClass {}
        protected class ProtectedClass {}
        protected internal class ProtectedInternalClass {}
        private class PrivateClass {}

        public AType<PublicClass, PublicClass> Foo1() {}
        protected AType<ProtectedClass, ProtectedInternalClass> Foo2() {}
        protected internal AType<PublicClass, ProtectedInternalClass> Foo3() {}
        internal AType<PublicClass, PublicClass> Foo4() {}
        private AType<PublicClass, PrivateClass> Foo5() {}

        public AType<ProtectedClass, PublicClass> Foo6() {}
        protected AType<ProtectedClass, PrivateClass> Foo7() {}
        protected internal AType<ProtectedClass, PublicClass> Foo8() {}
    }
```

For the most, there are no major surprises here. As you might expect, the "least" accessible type argument will determine the accessibility of each constructed type. Following this logic, you will find that the first five methods will all successfully compile. Method `Foo3()`, for example, is valid because both `protected` and `protected internal` type arguments conform to the accessibility of the method (which is `protected` in this case). The final three methods will all generate errors at compile time because each one violates these same accessibility rules. In reality, because private methods have no outside accessibility, they can be constructed with arguments that support every flavor of accessibility.

The following table provides a more complete breakdown of the accessibility rules that cover all the permutations of accessibility.

VB Accessibly	C# Accessibility	Valid Type Arguments
Public	public	public
Protected	protected	protected, protected friend/internal, public
protected friend	protected internal	protected internal, public

VB Accessibly	C# Accessibility	Valid Type Arguments
Friend	internal	internal, protected friend/internal, public
Private	private	public, protected, protected friend/internal, friend/internal, private

This list covers all the cases for methods declared within a class. However, these rules change somewhat if you declare your methods within the scope of an internal class. The following table calls out accessibility rules for an internal class.

VB Accessibly	C# Accessibility	Valid Type Arguments
Public	public	public, protected friend/internal, friend/internal
Protected	protected	protected, protected friend/internal, friend/internal, public
protected friend	protected internal	public, protected, protected internal, internal
Friend	internal	friend/ internal, protected friend/internal, public
Private	private	public, protected, protected friend/internal, friend/internal, private

The Default Keyword

When you're dealing with the implementation of a generic class, you may come across situations where you'd like to assign or access default values for your type parameters. However, because the actual type of your type parameter is unknown at the point of implementation, you have no way of knowing what's a valid default value for any given type parameter. To address this need, the generics specification introduced a new "default" keyword. This keyword was needed to allow you to determine the default value for generic types.

The language specification identifies a set of rules for the default values that will be returned for specific types of type parameters. These rules are as follows:

1. If a type parameter is s reference type, it will always return a default value of `null`.
2. If a type parameter is one of the built-in types, the default value will be assigned to whatever default is already defined for that type.
3. If a type parameter is a struct type, the default value will be the predefined default value for each of the fields defined in that struct.

This mechanism is essential in the implementation of some class types. Here's a simple example that demonstrates one application of the default keyword:

```
[C# code]
using System.Collections.Generic;

public class MyCache<K, V> {
```

```
    private Dictionary<K, V> _cache;
    public V LookupItem(K key) {
        V retVal;
        if (_cache.ContainsKey(key) == true)
            _cache.TryGetValue(key, out retVal);
        else
            retVal = default(V);
        return retVal;
    }
}
```

This example provides the shell of the implementation of a cache. It includes a `LookupItem()` method that looks for a specific key in the cache and returns a default value if it's not found. This is a basic application of the default mechanism. You could also use this feature to initialize the values of type parameters in advance of using them.

> **You'll notice that I did not provide an example of how to use the default keyword with Visual Basic. That's because Visual Basic does not currently support the default keyword in this context. In VB, the closet equivalent to this concept would be to set an instance of a type parameter equal to `nothing`.**

System.Nullable<T>

With the introduction of version 2.0 of the .NET Framework, developers are finally provided with a solution to the age-old problem of dealing with nullable types. The basic issue here is that not all data types provide a mechanism for determining if they have a "null" value. Clearly, with objects, there's a well-defined means of making this determination. However, with an `int` data type, there's no predefined value that could be used to determine if that int has been assigned a value. To resolve this, Visual Studio 2005 is introducing a new `Nullable` type that provides a uniform way of determining if a value is null.

Although nullable types are not exactly a generics concept, they are implemented using generics. A type is made nullable using the built-in `Nullable` generic class (which is in the `System` namespace). This generic class will be used to keep track of when its underlying type is assigned a value. Consider this example:

```
[VB code]
Public Class MyTest
    Public Shared Sub NullableTest(ByVal intVal1 As Nullable(Of Int32), _
                                   ByVal intVal2 As Int32)
        If (intVal1.HasValue() = True) Then
            Console.WriteLine(intVal1)
        Else
            Console.WriteLine("Value1 is NULL")
        End If

        If (intVal2 > 0) Then
            Console.WriteLine(intVal2)
         Else
```

```
            Console.WriteLine("Value2 is Null?")
         End If
    End Sub
End Class
```

```
[C# code]
using System;

public class MyTest {
    public static void NullableTest(Nullable<int> intVal1, int intVal2) {
        if (intVal1.HasValue == true)
            Console.WriteLine(intVal1);
        else
            Console.WriteLine("Value1 is NULL");

        if (intVal2 > 0)
            Console.WriteLine(intVal2);
        else
            Console.WriteLine("Value2 is Null?");
    }
}
```

This example declares a method that accepts two integer variables, one of which is nullable and one which is not. The body of this method then attempts to write out the value of each of these parameters to the console. Of course, you want your method to detect if either parameter has been actually assigned a value and only write that value out to the console. If it hasn't been assigned a value, you just dump a message indicating that no value exists. Simple enough.

Because the `intVal1` parameter was declared as a nullable type, you can easily detect if it is null by checking the `HasValue` property. Meanwhile, the `intVal2` parameter is never assigned a value, leaving you with no definitive means of determining if it's null. As a compromise, you could artificially decide that if it's greater than 0 it will be treated as non-null. However, that's an arbitrary rule you've defined in your code. Using the `Nullable` class gives you a universal, absolute definition for null that you can use throughout your code. It should be noted that the `Nullable` class only holds value types (int, double, and so on).

While the `Nullable<T>` type might look and behave like any other generic data container you'll find in the .NET framework, it is actually afforded special treatment by the CLR. As developers were initially trying out the `Nullable<T>` type, they discovered a few scenarios that yielded unexpected results. Consider the following example:

```
[VB code]
Dim intVal As New Nullable(Of Int32)
intVal = Nothing
Dim refVal As Object

refVal = intVal
If refVal Is Nothing Then
    Console.Out.WriteLine("Value is null")
End If
```

```
[C# code]
Nullable<int> intVal = null;
object refVal;

refVal = intVal;
if (refVal == null)
    Console.Out.WriteLine("Value is null");
```

This example declares a `Nullable<int>` instance before assigning that instance to `refVal`, which is an `object` data type. By making this assignment, you end up forcing your nullable `integer` type to be boxed. Now, if `Nullable<T>` were just another generic type, this boxing would have caused the null state of the value type to be lost during the boxing process. As you can imagine, this was not exactly the intended behavior.

To overcome this problem, the CLR was forced to make the nullable type a true runtime intrinsic. It was only at this level that runtime could provide behavior that would be more in line with what developers were expecting. So, as the CLR processes the un-boxing of the nullable types, it provides special handling to ensure that the null state of the value is not lost in translation.

By adding this capability, the CLR also added support for explicitly un-boxing a reference directly into a `Nullable<T>` type. The end result is a nullable type that is more directly supported by the CLR, which ultimately translates into a type that behaves much more intuitively.

> **C# provides an alternative syntax for declaring nullable types. By simply appending a `?` to your type (`int?`) you will have the equivalent of `Nullable<int>`. This mechanism provides developers with a shorthand way of declaring nullable types. While the declaration is certainly shorter, this syntax could be seen as impacting the readability of your code. In the end, it's more a matter of preference.**

Accessing Type Info

Now that you're actively building and consuming generic types, you might have an occasion when you'll want to access the specific type information for your generic types. Here's a simple example that dumps type information for a few generic classes:

```
[VB code]
Public Class OneParamType(Of T)
End Class

Public Class TwoParamType(Of T, U)
End Class

Public Class TypeDumper(Of T, U, V)
    Shared Sub DumpTypeInfo()
        Console.WriteLine(GetType(T))
        Console.WriteLine(GetType(U))
       Console.WriteLine(GetType(V))
        Console.WriteLine(GetType(OneParamType(Of String)))
        Console.WriteLine(GetType(OneParamType(Of T)))
        Console.WriteLine(GetType(TwoParamType(Of U, Integer)))
```

```
            Console.WriteLine(GetType(TwoParamType(Of T, V)))
        End Sub

        Public Sub ShowTypeInfo()
            TypeDumper(Of String, Integer, Double).DumpTypeInfo()
        End Sub
    End Class
```

```
    [C# code]
    using System;

    public class OneParamType<T> {}

    public class TwoParamType<T, U> {}

    public class TypeDumper<T, U, V> {
        public static void DumpTypeInfo() {
            Console.WriteLine(typeof(T));
            Console.WriteLine(typeof(U));
            Console.WriteLine(typeof(V));
            Console.WriteLine(typeof(OneParamType<String>));
            Console.WriteLine(typeof(OneParamType<T>));
            Console.WriteLine(typeof(TwoParamType<U, int>));
            Console.WriteLine(typeof(TwoParamType<T, V>));
        }

        public static void ShowTypeInfo() {
            TypeDumper<String, int, Double>.DumpTypeInfo();
        }
    }
```

This example creates a `TypeDumper` class that accepts three type arguments and includes a `DumpTypeInfo()` method that displays type information about each of these parameters in different contexts. Then, to see this method in action, the example includes a `ShowTypeInfo()` method that supplies `string`, `int`, and `double` type arguments. The output of calling this method will appear as follows:

```
    System.String
    System.Int32
    System.Double
    OneParamType`1[System.String]
    OneParamType`1[System.String]
    TwoParamType`2[System.Int32, System.Int32]
    TwoParamType`2[System.String, System.Double]
```

It's mostly what you'd expect. The one piece of information you might not have expected here is the number that appears after `OneParamType` and `TwoParamType`. That number represents the "arity" of the type, which corresponds to the number of type parameters that were used to construct the type.

Indexers, Properties, and Events

For the most part, generics represent a graceful extension to the languages of the .NET platform. That said, there are some areas of classes where generics cannot be applied. Specifically, generics cannot be applied to the indexers, properties, or events that appear in your classes. Each of these members can

reference type parameters in their signature. However, they are not allowed to directly accept type parameters. That distinction may not be clear. Following is a quick example that will help clarify this point. Let's start with an example that would be considered valid:

```
[VB code]
Imports System.Collections.Generic

Public Delegate Sub PersonEvent(Of T)(ByVal sender As Object, ByVal args As T)

Public Class Person(Of T)
    Private _children As List(Of T)

    Public Sub New()
        Me._children = new List(Of String)()
    End Sub

    Public Property Children() As List(Of T)
        Get
            Return Me._children
        End Get
        Set(ByVal value As List(Of T))
            Me._children = value
        End Set
    End Property

    Default Property Item(ByVal index As Long) As T
        Get
            Return Me._children(index)
        End Get
        Set(ByVal value As T)
            Me._children(index) = value
        End Set
    End Property


    Event itemEvent As PersonEvent(Of T)
End Class
```

```
[C# code]
using System.Collections.Generic;

public delegate void PersonEvent<T>(object sender, T args);

public class Person<T> {
    private List<T> _children;

    public Person() {
        this._children = new List<String>();
    }

    public List<T> Children {
        get { return this._children; }
        set { this._children = value; }
```

```
    }

    public T this[int index] {
        get { return this._children[index]; }
        set { this._children[index] = value; }
    }

    event PersonEvent<T> itemEvent;
}
```

You'll notice that this example includes references to its type parameter `T` in the declaration of a property, an indexer, and an event. In all of these cases, however, these members are referencing a type parameter that was supplied to the class. The difference is that none of these members can directly accept their own type parameters. So, the following would *not* be considered legal:

```
[VB code]
Public Class MySampleClass

    Public Sub New()
    End Sub

    Public ReadOnly Property Children(Of T)() As String
        Get
            ...
        End Get
    End Property

    Default Property Item(Of T)(ByVal index As Long) As String
        Get
            ...
        End Get
        Set(ByVal value As String)
            ...
        End Set
    End Property

    Event(Of T) itemEvent As SampleEvent
End Class
```

```
[C# code]
using System.Collections.Generic;

public class MySampleClass {

    public MySampleClass() {}

    public String Children<T> {
        get { ... }
        set { ... }
    }

    public String this<T>[int index] {
        get { ... }
```

```
        set { ... }
    }

    Event<T> SampleEvent itemEvent;
}
```

This is an example where the property, indexer, and event all accept their own type arguments. None of these forms of declarations will be deemed acceptable. Fortunately, this same constraint is not applied to methods.

Generic Structs

Classes and structs, for the most part, are synonymous. Essentially, whatever you can do with a class you can also do with a struct. Knowing this, you would be correct in assuming that all the same generic concepts you've seen applied to generic classes are also applicable to structs. Just to round things out, let's look at a simple example of a generic struct:

```
[VB code]
Imports System.Collections.Generic

Public Structure SampleStruct(Of T)
    Private _items As List(Of T)

    Public Function GetValue(ByVal index As Int32) As T
        Return Me._items(index)
    End Function

    Public Sub AddItem(ByVal value As T)
        Me._items.Add(value)
    End Sub

    Public Function ValidateItem(Of T)(ByVal value As Object) As Boolean
        Dim retVal As Boolean = False
        If (GetType(T).ToString.Equals(value)) Then
            retVal = True
        End If
        Return retVal
    End Function
End Structure
```

```
[C# code]
using System.Collections.Generic;

public struct SampleStruct<T> {
    private List<T> _items;

    public T GetValue(int index) {
        return this._items[index];
    }

    public void AddItem(T value) {
```

```
            this._items.Add(value);
        }

        public bool ValidateItem<T>(object value) {
            bool retVal = false;
            if (typeof(T).ToString() == value.ToString())
                retVal = true;
            return retVal;
        }
    }
```

The generic syntax you see here conforms, precisely, to the patterns that you've seen applied to generic classes. So, everything you've seen in this chapter regarding generic classes should be applied, universally, to generic structs.

Generic Interfaces

As part of looking at generic classes, it also makes sense to look at how generics can also be applied to the interfaces that are implemented by generic (or non-generic) classes. In many respects, generic interfaces actually conform to the same set of rules that govern the definition and usage of their non-generic counterparts. Here's a simple generic interface just to demonstrate the fundamentals of the syntax:

```
[VB code]
Public Interface SimpleInterface(Of T)
    Function IsValid(ByVal val As T) As Boolean
    Function GetValue() As T
    Function GetAllValues() As List(Of T)
End Interface

Public Interface ExtendedInterface(Of T)
    Inherits SimpleInterface(Of T)
    Sub Refresh()
End Interface

Public Class TestClass(Of T)
    Implements ExtendedInterface(Of T)
    . . .
    . . .
End Class
```

```
[C# code]
public interface SimpleInterface<T> {
    bool IsValid(T val);
    T GetValue();
    List<T> GetAllValues();
}
public interface ExtendedInterface<T> : SimpleInterface<T> {
```

```
    void Refresh();
}

public class TestClass<T> : ExtendedInterface<T> {
    . . .
    . . .
}
```

If you're already familiar with working with interfaces, this should be fairly trivial. You can see here that the interface accepts a type parameter that is then littered, in different forms, through the methods supplied by the interface. This also includes an example of generic interface inheritance so you can see type parameters used in that context. Finally, to make this complete, the example adds a class that implements that interface. There should be nothing particularly surprising here.

You should keep a few simple things in mind when working with generic interfaces. First, you should understand that each class that implements a generic interface can implement one and only one instance of generic interface. In the preceding example, suppose `TestClass` actually accepted two type parameters. In that scenario, it could not implement `ExtendedInterface<T>` and `ExtendedInterface<U>`. In this case, there would be instances where the compiler would not be able to resolve which method to call.

In some instances where you've implemented multiple interfaces, it may be necessary to qualify your method calls to be able to explicitly call out a method that's associated with a given interface. This can be achieved by simply pre-pending the interface declaration to a method.

Summary

All in all, you should come away from this chapter feeling like generic classes are not all that different their non-generic counterparts. Throughout this chapter, you have been exposed to each of the elements of a class and learned how generics are used to extend this model and make your classes more versatile, type-safe, and efficient. This included looking at constructors, inheritance, fields, methods, overloading, and all the constructs that are influenced by the introduction of generics. The chapter also looked at some of the rules that govern accessibility of generic classes. Finally, the chapter examined a series of other aspects of working with generic classes, including generic interfaces, generic structs, nullable generic types, and the default keyword. Equipped with this knowledge, you're likely to find plenty of new opportunities to leverage generic classes as part of your own solutions.