

1

Generics 101

For many programmers, generics will be an entirely new language feature. As such, it is important to establish a foundation of concepts that will clarify the role and significance of generics in the overall scheme of the .NET platform. This chapter provides this fundamental, conceptual view of generics that should provide you with a solid base of ideas that can be built upon in the chapters that follow. Along the way, you'll get the opportunity build your first generic types and get some exposure to the basic mechanics of generic types. This chapter also introduces a set of new terms that are used when referring to common generic concepts. You'll need to have a clear understanding of these terms because they are used throughout the book. Naturally, if you're already comfortable with the basics of generics, you may want to skip over this chapter.

Why Generics?

Most programmers can point to that one moment in their career where the light of abstraction or generalization went off in their head. If you have a background in structured programming, this might have been uncovered during a foray into the world of function pointers. Or, maybe it just occurred to you one day when you discovered you could extend the functionality of one of your methods by parameterizing some aspect of its behavior. If you're from the OO crowd, this probably happened one day when you stumbled upon your first real good use for polymorphism. At that moment, whenever it was, you realized that goal of generality, extensibility, and reusability that everyone had been evangelizing.

Now, with generics, you have an opportunity to wrap your brain around another form of generalization. This new brand of generalization will provide you with a host of new concepts to toss into your proverbial bag of coding and design techniques. And, once you've mastered generics, you may find yourself wondering how you lived without them for so long.

To understand the fundamental value of generics, you really need to see a compelling example. Let's start with a sample of some code that you might write without generics. Suppose you've

Chapter 1

decided to write your own Pyramid Manager product that will allow you to track the relationships between each of the salespeople in a pyramid scheme. You need to start with a basic domain object that will hold the common attributes of each of salesperson. The object is as follows:

[VB code]

```
Public Class SalesPerson
    Private _id As Integer
    Private _name As String

    Public Sub New(ByVal id As Integer, ByVal name As String)
        Me._id = id
        Me._name = name
    End Sub

    Public ReadOnly Property Id() As Integer
        Get
            Return Me._id
        End Get
    End Property

    Public ReadOnly Property Name() As String
        Get
            Return Me._name
        End Get
    End Property

    Public Overrides Function ToString() As String
        Return Me._name
    End Function
End Class
```

[C# code]

```
public class SalesPerson {
    private int _id;
    private string _name;

    public SalesPerson(int id, string name) {
        this._id = id;
        this._name = name;
    }

    public int Id {
        get { return this._id; }
    }

    public string Name {
        get { return this._name; }
    }

    public override string ToString() {
        return this._name;
    }
}
```

Now that you have an object to hold each salesperson, you'll want to place these salespeople into some form of hierarchical data structure where each `SalesPerson` can be associated with one or more "child" `SalesPerson` objects. Fortunately, you already have a tree structure that will let you represent just such a structure (this is an extremely simplified variant of the existing BCL `TreeNode`):

```
[VB code]
Imports System.Collections

Public Class TreeNode
    Private _nodeData As Object
    Private _childNodes As ArrayList

    Public Sub New(ByVal nodeData As Object)
        Me._nodeData = nodeData
        Me._childNodes = New ArrayList
    End Sub

    Public ReadOnly Property Data() As Object
        Get
            Return Me._nodeData
        End Get
    End Property

    Public ReadOnly Property Children() As TreeNode()
        Get
            Return Me._childNodes.ToArray()
        End Get
    End Property

    Default Public ReadOnly Property Item(ByVal index As Int32) As TreeNode
        Get
            Return Me._childNodes(index)
        End Get
    End Property

    Public Function AddChild(ByVal nodeData As Object) As TreeNode
        Dim newNode As New TreeNode(nodeData)
        Me._childNodes.Add(newNode)
        Return newNode
    End Function

    Overrides Function ToString() As String
        Return Me._nodeData.ToString()
    End Function
End Class
```

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

public class TreeNode {
    private object _nodeData;
    private ArrayList _childNodes;

    public TreeNode(object nodeData) {
```

```
        this._nodeData = nodeData;
        this._childNodes = new ArrayList();
    }

    public object Data {
        get { return this._nodeData; }
    }

    public TreeNode[] Children {
        get { return (TreeNode[])this._childNodes.ToArray(typeof(TreeNode)); }
    }

    public TreeNode this[int index] {
        get { return (TreeNode)this._childNodes[index]; }
    }

    public TreeNode AddChild(object nodeData) {
        TreeNode newNode = new TreeNode(nodeData);
        this._childNodes.Add(newNode);
        return newNode;
    }

    public override string ToString() {
        return this._nodeData.ToString();
    }
}
```

As you can see, your tree uses the least common denominator type of object to represent each of the items it manages. The result could certainly be considered a *generic* data container in that it can be populated with any data type. In this case, you're going to want to populate this structure with instances of your `SalesPerson` class. Here's some simple code that demonstrates how you would go about building a simple instance of your pyramid structure:

[VB code]

```
Dim rootNode, child1 As TreeNode
rootNode = New TreeNode(New SalesPerson(111, "Head Honcho"))
child1 = rootNode.AddChild(New SalesPerson(222, "Big Cheese"))
rootNode.AddChild(New SalesPerson(333, "Top Dog"))
child1.AddChild(New SalesPerson(444, "Big Enchilada"))
child1.AddChild(New SalesPerson(555, "Mr. Big"))
```

[C# code]

```
TreeNode rootNode = new TreeNode(new SalesPerson(111, "Head Honcho"));
TreeNode child1 = rootNode.AddChild(new SalesPerson(222, "Big Cheese"));
rootNode.AddChild(new SalesPerson(333, "Top Dog"));
child1.AddChild(new SalesPerson(444, "Big Enchilada"));
child1.AddChild(new SalesPerson(555, "Mr. Big"));
```

So, your pyramid is set now and you're ready to start rolling in the dough. But wait, as you might expect with any pyramid scheme, there's a catch. As you begin to work more intimately with your tree, you're going to discover that it has a significant flaw. Imagine a scenario where you've navigated to a specific

node in the tree and you want to access the information about the `SalesPerson` associated with that node in the tree. The code to retrieve the instance of the `SalesPerson` would appear as follows:

[VB code]

```
Dim aSalesPerson As SalesPerson
aSalesPerson = DirectCast(child1(0).Data, SalesPerson)
```

[C# code]

```
SalesPerson aSalesPerson = (SalesPerson)child[0].Data;
```

As you can see, in order to get your `SalesPerson` object out of the `TreeNode`, you are forced to cast the object type to the correct type. What's the big deal with that, you say? Well, if you value type safety in your code at all — and you should — you will find this cast a necessary evil that you'd much rather avoid. Later, in Chapter 2, "Valuing Type Safety," you see just how problematic this really is. If you're not already repulsed by this, you will be by the time you're more acclimated to generics.

The other downside you need to consider here, which may be even more significant, is the efficiency of this structure. Each time you put an item into your tree, it must be represented as an object. In this example, the `SalesPerson` was already an object, so no extra overhead was needed to make it conform to the requirements of your tree. However, to protect the innocent, let's assume you're going to eliminate the use of the `SalesPerson` type and simply populate the tree with sales totals for each person. So, your code to populate would be modified as follows:

[VB code]

```
Dim rootNode, child1 As TreeNode
rootNode = New TreeNode(3000.23)
child1 = rootNode.AddChild(1403.43)
rootNode.AddChild(943.94)
child1.AddChild(5123.94)
child1.AddChild(94994.0)
```

[C# code]

```
TreeNode rootNode = new TreeNode(3000.23);
TreeNode child1 = rootNode.AddChild(1403.43);
rootNode.AddChild(943.94);
child1.AddChild(5123.94);
child1.AddChild(94994.00);
```

In order to represent these numbers in your tree structure, each number will end up being boxed so it can be represented as an object. And, as you are probably aware, this process of boxing is going to introduce more overhead. This may seem negligible. However, you need to remember that containers of this nature can be populated with relatively large numbers of objects, which means any extra overhead associated with processing each item will impact performance exponentially.

At this stage, you'd probably agree that you'd like to overcome some of these shortcomings. What options are available to you? The typical solution to this dilemma is to create a type-safe version of the container that will support direct references to the type you want contained. For the Pyramid Manager example, creating a tree that accepts and returns `SalesPerson` objects would achieve this. Here's what the revised, type-safe version looks like:

Chapter 1

```
[VB code]
Imports System.Collections

Public Class SalesPersonNode
    Private _nodeData As SalesPerson
    Private _childNodes As ArrayList

    Public Sub New(ByVal nodeData As SalesPerson)
        Me._nodeData = nodeData
        Me._childNodes = New ArrayList
    End Sub

    Public ReadOnly Property Data() As SalesPerson
        Get
            Return Me._nodeData
        End Get
    End Property

    Public ReadOnly Property Children() As Array
        Get
            Return Me._childNodes.ToArray()
        End Get
    End Property

    Default Public ReadOnly Property Item(ByVal index As Long) As SalesPersonNode
        Get
            Return Me._childNodes(index)
        End Get
    End Property

    Public Function AddChild(ByVal nodeData As SalesPerson) As SalesPersonNode
        Dim newNode As SalesPersonNode = New SalesPersonNode(nodeData)
        Me._childNodes.Add(newNode)
        Return newNode
    End Function

    Overrides Function ToString() As String
        Return Me._nodeData.ToString()
    End Function
End Class
```

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

public class SalesPersonNode {
    private SalesPerson _nodeData;
    private ArrayList _childNodes;

    public SalesPersonNode(SalesPerson nodeData) {
        this._nodeData = nodeData;
        this._childNodes = new ArrayList();
    }

    public SalesPerson Data {
```

```

        get { return this._nodeData; }
    }

    public SalesPerson[] Children {
        get {
            return (SalesPerson[])this._childNodes.ToArray(typeof(SalesPerson));
        }
    }

    public SalesPerson this[int index] {
        get { return (SalesPerson)this._childNodes[index]; }
    }

    public SalesPersonNode AddChild(SalesPerson nodeData) {
        SalesPersonNode newNode = new SalesPersonNode(nodeData);
        this._childNodes.Add(newNode);
        return newNode;
    }

    public override string ToString() {
        return this._nodeData.ToString();
    }
}

```

This new `SalesPersonNode` gives you a very type-safe approach to building your pyramid. It also allows you to introduce more domain-specific operations to the collection without fear of breaking its generality. The biggest problem with this, though, is that it forces you to create a separate class for every data type you want to contain. For example, if you want to go back to the previous example where the tree held only numbers, you'd need to make a `DoubleTreeNode`. And, for the most part, that's what developers have often done. They essentially end up bloating their overall code size to support each of these type-specific structures. Or, they've ended up living with some of the downside of the less type-safe solutions (blech). Neither approach is all that appealing.

Enter Generics

As you can imagine by now, the problems pointed out in these examples are at the very core of the rationale for introducing generics. With generics, you can finally strike a balance between type safety and generality while, at the same time, eliminating the need to overpopulate your libraries with a gaggle of unnecessary classes. Let's look at how generics would be applied to the Pyramid Manager example. Making this change will mostly involve rewriting the `TreeNode` class. The `SalesPerson` object will remain unscathed as part of this conversion.

```

[VB code]
Imports System.Collections

Public Class TreeNode(Of T)
    Private _nodeData As T
    Private _childNodes As ArrayList

    Public Sub New(ByVal nodeData As T)
        Me._nodeData = nodeData
        Me._childNodes = New ArrayList
    End Sub

```

Chapter 1

```
End Sub

Public ReadOnly Property Data() As T
    Get
        Return Me._nodeData
    End Get
End Property

Public ReadOnly Property Children() As TreeNode(Of T)()
    Get
        Return Me._childNodes.ToArray()
    End Get
End Property

Default Public ReadOnly Property Item(ByVal index As Long) As TreeNode(Of T)
    Get
        Return Me._childNodes(index)
    End Get
End Property

Public Function AddChild(ByVal nodeData As T) As TreeNode(Of T)
    Dim newNode As TreeNode(Of T) = New TreeNode(Of T)(nodeData)
    Me._childNodes.Add(newNode)
    Return newNode
End Function

Overrides Function ToString() As String
    Return Me._nodeData.ToString()
End Function

End Class
```

[C# code]

```
using System.Collections;

public class TreeNode<T> {
    private T _nodeData;
    private ArrayList _childNodes;

    public TreeNode(T nodeData) {
        this._nodeData = nodeData;
        this._childNodes = new ArrayList();
    }

    public T Data {
        get { return this._nodeData; }
    }

    public TreeNode<T>[] Children {
        get { return (TreeNode<T>[])this._childNodes.ToArray(typeof(TreeNode<T>)); }
    }

    public TreeNode<T> this[int index] {
        get { return (TreeNode<T>)this._childNodes[index]; }
    }
}
```

```

    }

    public TreeNode<T> AddChild(T nodeData) {
        TreeNode<T> newNode = new TreeNode<T>(nodeData);
        this._childNodes.Add(newNode);
        return newNode;
    }

    public override string ToString() {
        return this._nodeData.ToString();
    }
}

```

The first thing you should notice here is how the declaration of `TreeNode` changed. The class name now has some additional information appended to it, which indicates that it is a generic type. As such, your `TreeNode` will now accept a type as a parameter. This means that you can officially abandon your need to cling to the `object` data type as the means of genericizing your `TreeNode`. Instead, you can use this incoming parameter `T` to represent the specific type of object that will be contained by your tree node.

The other change you'll notice is that all the references to the `object` data type have been replaced with a type parameter, `T`. In reality, as you look at this class now, it doesn't seem all that different from the non-generic version. You've essentially just modified it to accept a parameter that is used as a placeholder for the data type that will end up being substituted at run-time.

Now that you have a new generic type, you need to figure out how to populate it with data. As a consumer of a generic type, you should find that working with a generic type doesn't introduce any significant new concepts. Mostly, you just need to provide the additional type parameter to the class when you declare each new instance of the `TreeNode`. This will provide the compiler and the CLR all the information they need to successfully construct the run-time representation of your generic class. Here's the generic version of the code that is used to populate the tree:

```

[VB code]
Dim rootNode, child1 As TreeNode(Of SalesPerson)
rootNode = New TreeNode(Of SalesPerson)(New SalesPerson(111, "Head Honcho"))
child1 = rootNode.AddChild(New SalesPerson(222, "Big Cheese"))
rootNode.AddChild(New SalesPerson(333, "Top Dog"))
child1.AddChild(New SalesPerson(444, "Big Enchilada"))
child1.AddChild(New SalesPerson(555, "Mr. Big"))

```

```

[C# code]
TreeNode<SalesPerson> rootNode =
    new TreeNode<SalesPerson>(new SalesPerson(111, "Head Honcho"));
TreeNode<SalesPerson> child1 =
    rootNode.AddChild(new SalesPerson(222, "Big Cheese"));
rootNode.AddChild(new SalesPerson(333, "Top Dog"));
child1.AddChild(new SalesPerson(444, "Big Enchilada"));
child1.AddChild(new SalesPerson(555, "Mr. Big"));

```

Notice that, as each `TreeNode` is constructed, it must be provided with a data type. In this example, a `SalesPerson` data type is provided, which then forces all references to the type parameter `T` to be replaced, at run-time, with the type `SalesPerson`. And, as you access the contents of your tree, it's able

Chapter 1

to return you these `SalesPerson` instances without requiring those unappealing casts that you were forced to employ in the previous example.

Although this example only provides a small glimpse into the functionality of a generic type, it should make it apparent that generics are going to find their way into your code. With generics, I cannot imagine any scenario where you'd ever want to compromise and use any kind of data container that wasn't type safe.

Hello Generics

Every programming book known to man seems to include the obligatory "Hello World" example. It's the de facto standard that is used to provide a quick, minimal glimpse of a functioning program. And, with the generics rationale out of the way, it only seems fair to offer up my own "Hello Generics" example that takes the classic version and spruces it up with a slight generics twist. This example will also give you another opportunity to see generics in action.

[VB code]

```
Public Class HelloGenerics(Of T)
    Private _thisTalker As T

    Public Property Talker() As T
        Get
            Return Talker
        End Get
        Set(ByVal value As T)
            Me._thisTalker = value
        End Set
    End Property

    Public Sub SayHello()
        Dim helloWorld As String
        helloWorld = _thisTalker.ToString()
        Console.WriteLine(helloWorld)
    End Sub
End Class
```

[C# code]

```
public class HelloGenerics<T> {
    private T _thisTalker;

    public T Talker {
        get { return this._thisTalker; }
        set { this._thisTalker = value; }
    }

    public void SayHello() {
        string helloWorld = _thisTalker.ToString();
        Console.WriteLine(helloWorld);
    }
}
```

The first step is to create a new generic type, `HelloGenerics`, that accepts a single type parameter. The idea here is to build a generic type can accept any object type and ask it to say “Hello World”. So, instead of having the limitation of saying hello in a single language, the generic type is going to be used to provide a more dynamic, more worldly solution that says hello in a variety of tongues. After all, if it’s going to say hello to the world, it should not expect that everyone is going to understand English.

The next step is to create a pool of objects that can be passed as parameters to the `HelloGenerics` type, each with its own language-specific variation on how to say hello. You should notice that these can be objects of any type and they are not required to share any common base class that provides a virtual interface for saying hello. Although valid, that would be the pure OO way to do this and would not demonstrate the generic approach to this problem. The lineup of international objects is as follows:

```
[VB code]
Public Class GermanSpeaker
    Public Overrides Function ToString() As String
        Return "Hallo Welt!"
    End Function
End Class

Public Class SpanishSpeaker
    Public Overrides Function ToString() As String
        Return "Hola Mundo!"
    End Function
End Class

Public Class EnglishSpeaker
    Public Overrides Function ToString() As String
        Return "Hello World!"
    End Function
End Class

Public Class APLSpeaker
    Public Overrides Function ToString() As String
        Return "!dlroW olleH"
    End Function
End Class
```

```
[C# code]
public class GermanSpeaker {
    public override string ToString() {
        return "Hallo Welt!";
    }
}

public class SpanishSpeaker {
    public override string ToString() {
        return "Hola Mundo!";
    }
}

public class EnglishSpeaker {
    public override string ToString() {
        return "Hello World!";
    }
}
```

```
    }  
}  
  
public class APLSpeaker {  
    public override string ToString() {  
        return "!dlroW olleH";  
    }  
}
```

Two random observations stood out after I put these classes together. First, I fully expected the German version of this to be much longer. Every international translation of software I ever worked on for Germany seemed to double the length of every string. Also, I tossed APL in here because, as a programming language, it always seemed foreign to me.

The next step in this process is to get this generic type actually speaking. This is accomplished by constructing a few instances of `HelloGenerics`. The following code will take care of this last bit of work:

```
[VB code]  
Dim talker1 As New HelloGenerics(Of GermanSpeaker) ()  
talker1.Talker = New GermanSpeaker()  
talker1.SayHello()  
  
Dim talker2 As New HelloGenerics(Of SpanishSpeaker) ()  
talker2.Talker = New SpanishSpeaker()  
talker2.SayHello()
```

```
[C# code]  
HelloGenerics<GermanSpeaker> talker1 = new HelloGenerics<GermanSpeaker> ();  
talker1.Talker = new GermanSpeaker();  
talker1.SayHello();  
  
HelloGenerics<SpanishSpeaker> talker2 = new HelloGenerics<SpanishSpeaker> ();  
talker2.Talker = new SpanishSpeaker();  
talker2.SayHello();
```

All that's left now is to run this code and you'll see the multilingual "Hello World" break through all new international barriers. Although not all that practical (what "hello world" app is?), this example does help to clarify the basic steps that are involved in building and consuming a simple generic type.

A More Conceptual View

At this stage, it's my hope that you have a much better feeling for why the term *generics* was coined to describe this language feature. Generics bring a new level of generality to your types, which allows you to separate the behavior of a class from the data types that it operates on. This is, in essence, precisely what makes the type generic. Through generics, you are able to add parameters to your types much like you would add parameters to your methods to extend their generality. And, just as parameters for your methods allow you to alter the nature of your method, so too do generic type parameters allow you to alter the representation of your classes, methods, and so on.

The beauty of generics, as you see in more detail in the ensuing chapters, is that this mechanism allows you to build more adaptable, more general versions of your code. Your classes, methods, and interfaces

are able to take on this new dimension of generality while still allowing you to write more robust, more type-safe code. The truth is, the type-safety benefits — on their own — make generics worth the cost of admission.

So, as you begin to work with generics, you should try to be more than a consumer of the standard generic types. You should look for opportunities to construct your own generic types, introduce generic methods, or leverage any number of the generic mechanisms that are covered in the scope of this book. Once you get comfortable with the concepts, you're likely to find yourself infusing generics into your approach to a much wider spectrum of solutions than you may have initially envisioned.

Parametric Polymorphism

While I'm being conceptual, it's important to introduce the idea of *parametric polymorphism*. The term is often used to describe the flavor of polymorphism that can be achieved with generic types. To understand the concept, let's turn back the time machine and look at the classic example that is used to convey the root concept of polymorphism. The diagram in Figure 1-1 shows a basic object hierarchy with a `Shape` base class and a series of specialized shape types.

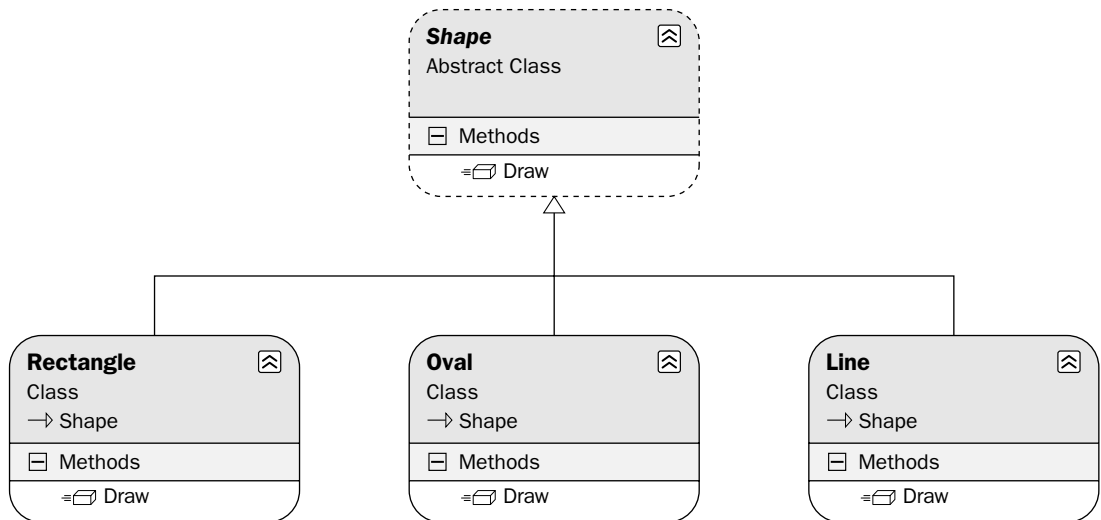


Figure 1-1

This example demonstrates how behavior can be generalized to a base class (`Shape`) and, through polymorphism, provides a specific implementation of a `Draw` method for each type of `Shape`. The beauty of polymorphism is that you can introduce new, specialized behaviors for a `Shape` without altering anything about the client's fundamental view of a `Shape`. If you decide you want to add a `Square`, you can add it and it will immediately be on equal footing with any other `Shape` in the system.

This little trip down polymorphic memory lane illustrates the fundamental idea behind polymorphism. So, what is parametric polymorphism? Well, instead of achieving polymorphism through inheritance, generics allow you to achieve the functional equivalent by allowing you to parameterize your types. Where regular polymorphism might use a virtual method table to override the methods of a parent object, parametric polymorphism achieves a similar result by allowing a single class to dynamically

Chapter 1

substitute the types referenced in its internal implementation. This ability to alter a class's behavior via a type parameter is seen simply as an alternative form of polymorphism, thus the name parametric polymorphism.

While I think it would be incomplete to discuss generics without including parametric polymorphism, it's also fair to say that the .NET implementation of generics imposes some constraints that limit the amount of polymorphic behavior it can achieve. C++ and other compile-time approaches to generics, as discussed in Chapter 3, "Generics ≠ Templates," provide developers with a richer set of polymorphic possibilities.

Terminology

In addition to nailing down generic concepts, it's important to establish a clear set of terms that are used to describe the different facets of generics. It's also important for you to have some precision in your generic vocabulary, because many of these terms are referenced heavily throughout the remainder of this book.

First, I'll start by building the shell of a simple generic type that can be referenced as part of this exploration of generic terminology. The following generic `Stack` type should serve that purpose well:

```
[VB.NET Example]
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# Example]
public class Stack<T> {
    private T[] items;
    private int count;

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

Type Parameters

A type parameter refers to the parameter that is used in the definition of your generic type. In the `Stack` example, the class accepts one type parameter, `T`. Each generic type can accept one or more type parameters, and this list of parameters will define the signature of your type. The names used for these parameters are then referenced throughout the implementation of your new type. For the `Stack`, you can see where multiple references have been added to the `Stack`'s type parameter, `T`.

Although type parameters can be applied to classes, structs, and interfaces, they cannot be directly applied to indexers, properties, or events. So, when you invoke a property of an object, for example, you cannot supply any type arguments. This is not to say that these constructs have no awareness of type parameters. Indexers, properties, and events can all reference type parameters in their signatures; they simply can't explicitly accept their own type arguments. Instead, those types must be defined as part of the surrounding class.

Open Types

Although `Stack` shares many of the characteristics of any class you might declare, its ability to accept a type parameter as part of its declaration means you need to further qualify the existing naming convention to accurately describe this new construct. Instead of referring to this as a class, generics consider the `Stack<T>` an "open type." My assumption here is that the term "open" is meant to convey the idea that the type is not fully defined and is "open" to taking on multiple concrete representations. If you have some exposure to C++ templates, you may be more comfortable referring to this as a parameterized type. For the sake of this discussion, though, we will stick with the accepted .NET generics terminology.

Constructed Types

Open types and type parameters are all about defining the structure of your generic type. A constructed type, on the other hand, represents a concrete instance of one of your open types. To create a constructed type from your open `Stack` type, you'd execute the following code:

```
[VB code]
Dim myStringStack As New Stack(Of String)
```

```
[C# code]
Stack<string> myStringStack;
```

This constructed type shares many of the attributes of traditional .NET types. They do, however, have some distinguishing syntactic characteristics worth exploring.

Type Arguments

Type arguments are likely the simplest concept to explain. Whenever you instantiate a constructed type, you must provide specific types for each of the type parameters required by the given open type you are constructing. So, when you declared the constructed type `Stack<string>` in the preceding section, the `string` type passed in would be considered a type argument.

Open and Closed Constructed Types

When creating a constructed type, you are not always required to provide a type argument. Take a look at the following snippet of a generic type declaration, which illustrates a scenario where this would be valid:

```
[VB.NET Example]
Public Class MyType(Of T)
    Private constructedType1 As MyOtherType(Of Integer)
    Private constructedType2 As MyOtherType(Of T)
    ...
End Class
```

```
[C# Example]
public class MyType<T> {
    private constructedType1<Integer> member1;
    private constructedType2<T> member2;
    ...
}
```

This example creates an open type `MyType`, which has two data members that are constructed types. The first data member, `constructedType1`, is considered a *closed* constructed type because its type argument is fixed or “closed” to further definition. Its type argument will always be an `Integer`. The other data member, `constructedType2`, throws in a new twist. Instead of passing a concrete type as its type argument, it passes a type argument of `T`, which is the type parameter defined for the generic type. Despite this variation, the type is still considered a constructed type. However, because its parameter is still open to run-time definition, it is referred to as an open constructed type.

Generic Methods

So far, the examples of generic types have been limited to generic classes. However, generics can also be applied to individual methods. The following is a very simple example of a generic method:

```
[VB.NET Example]
Public Function CalculateValue(Of T)(myParam1 as T, myParam2 as Integer) As T
    Dim var1 As T
    ...
End Function
```

```
[C# Example]
public T CalculateValue<T>(T myParam1, int myParam2) {
    T var1;
    ...
}
```

Like open types, generic methods also accept a type parameter. And, like open types, generic methods can reference this parameter as part of their signature or implementation. In fact, this example loads up the references to the type parameter to illustrate this point. The return type, one of its parameters, and a local variable all reference the type parameter `T`. Chapter 5, “Generic Methods,” looks more closely at the details of generic methods.

Type Instantiation

The first time the Just-In-Time (JIT) compiler comes across a constructed type in your code, it must transform that type into the appropriate IL representation. During this process, it will examine each of the incoming type arguments and substitute each of the open type’s parameters with the data types of these

arguments. The result will be the accurate run-time representation of your constructed type. This transformation process is considered “type instantiation” because it yields an actual instance of the constructed type.

Arity

Arity simply refers to the number of type parameters that are used by a generic type. So, if your type has three type parameters, it is said to have an arity of 3. And, just to be complete, a type that has no type parameters has an arity of zero. You’ll often find this term used in scenarios where you are using reflection to examine the characteristics of generic types.

Generic Types

Generic types is probably the most heavily used term referenced throughout this book. It is the all-encompassing term that is intended to describe any class, struct, event, or delegate that accepts one or more type parameters. This term is not really part of any formally accepted generics terminology and is somewhat synonymous with the idea of an open type. However, it’s often a more useful term to invoke when attempting to describe the broadest definition of any type that supports generic behavior.

Bringing It All Together

So, now that you’re equipped with generics terminology, let’s take it out for a spin. When you declare a generic type, that type is referred to as an open type (`MyType<T>`). And, when you declare an instance of that type by passing type arguments into the type parameters (`MyType<int>`), you form a constructed type. Whew. That was a mouthful. Still, it’s just that kind of phrasing that’s sure to make you a hit at the next Christmas party.

Summary

The goal of this chapter was to introduce the basic concepts associated with constructing and consuming generic types. It started out by examining a simple non-generic solution. As part of that example, you looked at some of the pitfalls that are typically associated with using these non-generic types. With that as a backdrop, you then went on to explore how generics can be applied to overcome the issues that are highlighted in these examples. The chapter also touched, briefly, on some of the higher-level conceptual aspects of generics, which should give you a better idea of how generics concepts can be applied when constructing your own types. To round things out, the chapter also included a “Hello Generics” example that provides a generic version of the traditional “Hello World” application. Finally, the chapter wrapped up with a look at generic terminology. Understanding this terminology is fundamental to concepts that appear throughout this book. Collectively, these topics should give you a reasonable starting point for learning more about generics.

