10

Generics Guidelines

With each significant new language feature also comes a set of guidelines that dictate how and when that feature should be applied. Generics are no different. This chapter assembles a set of guidelines that attempt to address some of the common practices that should be applied or given consideration when consuming or constructing generic types. As part of this effort, it provides an item-by-item breakdown of the guidelines and, where necessary, digs into the pros and cons associated with a given guideline. The goal here is to bring together, in one place, all those generics practices that are being discussed, debated, and adopted by the development community.

An Evolving List

Although generics can't be classified as *new,* they certainly will be showing up on the desktops of a whole new group of programmers with the release of Visual Studio 2005. With generics being unfolded to a broader audience and with .NET introducing its own new variations on the generics theme, it's easy to see why generics require the introduction of some new guidelines. It's also fair to assume that this list of guidelines is very much in its infancy. Once developers are using generics in full force, I would expect the list of generics guidelines to continue to grow and mature.

To kick-start this process, Microsoft has been assembling a preliminary list of items that shape much of the thinking on guidelines at this stage. The goal of this chapter is to distill that list, add new items, and generally provide a more thorough examination of rationale behind applying these guidelines. Overall, this effort will produce a more formalized look at the factors that are likely to influence some aspects of your generic thinking.

Defining Guidelines

As guidelines, I would also expect there to be some level of disagreement about these items. If you can't get programmers to agree on tabs versus spaces, you're certainly not going to get them to

reach consensus on areas that have even higher levels of grey matter. So, as you review this list, you need to keep in mind that these are only guidelines and are not being represented as rules that are set in stone. Guidelines can and *should* be violated under certain circumstances. They exist purely to help you define the rules that should shape your general process for deciding how and when to use a generic type. When you find exceptions to the rule, by all means—violate the rule. Just be sure that you can defend each violation and, if you can, you'll be fulfilling the spirit of what the guideline is trying to achieve.

Organization

In general, when you're assembling a list of guidelines, they don't always fall into natural categories. However, as I looked at the list I had, I did see some items fitting into specific clusters that seemed to conform to a specific theme. Within each cluster, I simply list each item with a number, which provides me with a simple mechanism for referencing each item individually.

Identifying Generic Opportunities

This first set of guidelines is focused on describing a specific set of scenarios where you should consider leveraging generics. These items represent areas where you would want to consider refactoring existing code or they may just be patterns you'll want to consider when you're introducing new code. Some of these may be somewhat obvious based on other topics covered elsewhere in this book. However, the goal is to assemble all of these items in one place as a list that you can easily consult as you're working with generics.

Item 1: Use Generic Collections

Data collections are typically one of the most heavily used data types. You likely already have `ArrayList`s and `HashTable`s strewn throughout your existing code. You were also likely—before generics—to make heavy use of these `System.Collections` data structures in new code you would be writing. However, with generics, there's really no good reason to continue to use the collections from this namespace.

If there is one area where generics add unquestionable value, it is in the area of collections. Without generics, producers and consumers of non-generic collections were forced to represent contained types as objects. This, of course, meant your code was littered with casts and general type coercion to covert each object to its actual type. It also meant that value types needed to be boxed to be represented as object types. Even in cases where you may have tried to limit the impact of non-generic collections, you were still typically forced to bloat your code with type-specific collection wrappers. For these reasons and a hundred others sprinkled throughout this book, it should be clear that there are few compelling reasons to cling to these old, non-generic collections. In fact, I would argue that generic collections represent the single most compelling usage of generics and, if you're not sold on the value of using generic collections, you're not likely to be sold on *any* of the value generics can bring to your code.

Although I think the arguments for using generic collections are compelling, not every solution may have the luxury of fully replacing non-generic collections with their generic counterparts. If you expose a public API and have clients that currently bind to that non-generic API, you're going to need to figure out how to transition your API to generics. In these instances, it would still seem valuable to leverage generic collections within your implementation and, over time, ease generics into your API.

Item 2: Replace Objects with Type Parameters

Before generics, programmers in search of generality typically found themselves relying on the `object` type as the universal solution to achieving generality. If you had a class or methods that had common functionality that could be applied to disparate types, you had few options at your disposal. If you didn't have a common base class or interface, your only alternative was to use a least common denominator type, the `object` type. For example, suppose you had the following method to send messages:

```
[VB code]
Public Function SendMsg(ByVal sender As Object, ByVal param As Object) as Object
End Function

[C# code]
public object SendMsg(object sender, object param) {}
```

This method provides a very general-purpose mechanism for sending a message from any `object` type with any parameter type and any return type. By using the `object` type throughout this method, you've allowed this method to be used with a wide spectrum of types. Of course, you've also completely traded off type safety for generality here.

As you can imagine, generics are a perfect fit for solving the type-safety issues introduced by this method. Through generics, you can strike a balance between type safety and generality, which is exactly what you're looking for in this scenario. The generic version of this method would appear as follows:

```
[VB code]
Public Function SendMsg(Of I, J, K)(ByVal sender As I, ByVal param As J) as K
End Function

[C# code]
public K SendMsg<I, J, K>(I sender, J param) {}
```

You can see here that the `SendMsg()` method has been converted into a generic method that uses type parameters in each of the slots where it had previously used `object` types.

This example illustrates just one instance where `object` types can be made type-safe through the use of generics. You may be using `object` data types in a variety of different contexts and, for each of those, you should be considering swapping out these `object` types with some flavor of generic solution.

The basic rule of thumb here is that, with generics, there should be a much lower incidence of `object` types showing up in your code. Wherever you spot an `object` you should be asking yourself if generics can be applied to eliminate the dependency on this `object` type. Generics should make least common denominator programming the anomaly instead of the norm.

Item 3: Replace System.Type with Type Parameters

In some instances, you may have used references to `System.Type` in the signature of your methods, allowing you to alter the behavior of your method based on a supplied type. For example, it wouldn't be all that uncommon in the pre-generic era to find a method that used a type parameter as follows:

```
[VB code]
Public Function FindPerson(ByVal personType As Type, ByVal Int32 As id) As Object
End Function
```

```
[C# code]
public object FindPerson(Type personType, int id) {}
```

This method takes a `System.Type` type as an incoming parameter and searches for people that have an `id` that matches the supplied `id`. If it finds a match, it will construct an object that corresponds to the supplied type (`Customer`, `Employee`, and so on) and return that as the output of this function call. This method might come in handy in scenarios where you have specialized `Person` objects, each of which has a unique `id`. It allows you to find and construct any descendant `Person` type without requiring separate methods to support each type.

Before generics, this would not have been an unreasonable piece of code to find. However, with generics, you shouldn't find yourself needing to rely on the `System.Type` nearly as much. In fact, this method could be made much cleaner by making it a generic method and retrofitting it with type parameters as follows:

```
[VB code]
Public Function FindPerson(Of T)(ByVal personType As T, ByVal Int32 As id) As T
End Function

[C# code]
public T FindPerson<T>(T personType, int id) {}
```

This makes for a cleaner interface and likely reduces the complexity of this method's implementation. It also means that consumers of this method won't be forced to cast this method's return value to a specific type.

Item 4: Use Type Parameters for Ref Types (C# Only)

Item 2 talked about the general strategy of replacing object types with type parameters. There is one variant of this rule that seems relevant enough to warrant the introduction of a new item. For this item, the focus is on the use of `object` data `types` as reference parameters. With C#, a reference parameter will only accept references that match, exactly, the type identified by the reference parameter. Consider, for example, the following method that accepts a reference parameter:

```
[C# code]
public void Sort(ref object param1, ref object param2) {}
```

This method was created to sort objects of any type. And, as such, it took the least common denominator approach of using an `object` type as the reference it accepts for its two parameters. The fact that these two parameters are identified as `object` types wouldn't seem like a real problem. Here's a look at what happens when you construct two `Person` objects and call this `Sort()` method:

```
[C# code]
public void processItems() {
    Person person1 = new Person(424);
    Person person2 = new Person(190);

    Sort(ref person1, ref person2);
}
```

On the surface, this would appear to be fine. The `Person` objects, which are rooted in `object`, will simply get cast to an `object` and passed successfully as parameters to this method. And, if this method didn't

specify these parameters as reference types, that logic would be fine. However, as mentioned earlier, with reference types the compiler will require the supplied parameters to match the precise type that is called out in the signature of the method. And, in this example, `Person` will not match `object`.

Now, you could solve this with a handy dandy cast, casting each `Person` object to an `object` type. However, that's not necessary. You can resolve this problem by making your `Sort()` method generic and using type parameters in place of the `object` types. The new version would appear as follows:

```
[C# code]
public void Sort<T>(ref T param1, ref T param2) {}
```

This change makes your types match exactly and, because types can be inferred, the previous sample client code for this method can remain untouched.

VB seems to handle this scenario more gracefully. It does not appear to require the incoming types to match the precise signature of the types declared in the method. Still, even with VB, you should see that it still makes sense to use a generic method here. In the spirit of Item 2, you should still be looking for opportunities to rid your code of `object` types.

In many respects, this rule may appear to be a duplicate of Item 2. And, in the general sense, it is a duplicate. However, the added twist associated with using reference types seems to stand out as one more variation that's worth considering in isolation.

Item 5: Genericize Types That Vary Only by a Data Type

If you look across all of your existing classes, interfaces, delegates, and methods, you are likely to identify code that varies primarily by the types it contains and/or manages. In these cases, you need to consider whether generics can be applied, allowing a single implementation to service the needs of multiple data types. Applying generics in these scenarios can produce a variety of positive side effects, including reducing code size, improving type safety, and so on. The following sections provide examples of some of these refactoring themes.

Eliminating Redundant Data Containers

By far, data containers represent one of the most common, straightforward areas where you will want to do some generic refactoring. Most solutions have at least one or two examples where, in your distaste for the compromised type safety of an `ArrayList`, you created your own type-safe wrapper classes. It would not be uncommon, for example, to find pre-generic code that might appear as follows:

```
[VB code]
Public Class PersonCollection
    Private _persons As ArrayList

    Public Sub New()
        _persons = New ArrayList()
    End Sub

    Public Sub Add(ByVal person As Person)
        _persons.Add(person)
    End Sub
```

```
    Public ReadOnly Property Item(ByVal Index As Int32) As Person
        Get
            Return DirectCast(_persons(Index), Person)
        End Get
    End Property
End Class

Public Class OrderCollection
    Private _orders As ArrayList

    Public Sub New()
        _orders = New ArrayList()
    End Sub

    Public Sub Add(ByVal Order As Order)
        _orders.Add(Order)
    End Sub

    Public ReadOnly Property Item(ByVal Index As Int32) As Order
        Get
            Return DirectCast(_orders(Index), Order)
        End Get
    End Property
End Class
```

```
[C# code]
public class PersonCollection {
    private ArrayList _persons;

    public PersonCollection() {
        _persons = new ArrayList();
    }

    public void Add(Person person) {
        _persons.Add(person);
    }

    public Person this[int index] {
        get { return (Person)_persons[index]; }
    }
}

public class OrderCollection {
    private ArrayList _orders;

    public OrderCollection() {
        _orders = new ArrayList();
    }

    public void Add(Order order) {
        _orders.Add(order);
    }
```

```
        public Order this[int index] {
            get { return (Order)_orders[index]; }
        }
    }
```

These two classes wrap an `ArrayList` and expose a type-safe interface that shields clients from the object-reality that comes along with using a non-generic container. After looking at these two classes, it should be obvious that they are perfect candidates for generic refactoring. And, while each of these classes can be improved through the application of generics, there are broader issues to consider here.

Outside of the data types being managed here, `Person` and `Order`, there's nothing different in their actual implementation. And, whenever this is the case, you know you have a situation that is crying out for the application of generics. In fact, the `Collection<T>` container that is provided as part of `System.Collections.Generic` would eliminate the need for any of this code to exist. By simply declaring `Collection<Person>` and `Collection<Order>`, you would get all the type safety and functionality that's provided in the preceding example.

If your collection classes had introduced functionality that was not directly supported by `Collection<T>`, you would still simply create a new class that subclassed `Collection<T>` and added any new custom members.

This example is straight out of the Generics 101 bible. As such, it may have already been evident. Still, to overlook this scenario in the context of generic guidelines would be a mistake — especially because this should be one of the areas where generics will deliver the most value.

Identifying Candidate Methods

Finding methods that are candidates for generic refactoring is a more subtle, less exact science. The fundamentals are still the same. You essentially want to look for sets of methods that vary, primarily by the data type they are processing. The most common examples that seem to show up here are those methods that perform very basic operations on whole objects without calling specific methods. The following examples fall into this category:

```
[VB code]
Public Shared Sub Swap(ByRef val1 As String, ByRef val2 As String)
    Dim tmpObj As String = val2
    val2 = val1
    val1 = tmpObj
End Sub

Public Shared Sub Swap(ByRef val1 As Double, ByRef val2 As Double)
    Dim tmpObj As Double = val2
    val2 = val1
    val1 = tmpObj
End Sub

Public Shared Function Max(ByVal val1 As Int32, ByVal val2 As Int32) As Int32
    Dim retVal As Int32 = val1
    If (val2 > val1) Then
        retVal = val2
    End If
    Return retVal
End Function
```

```
Public Shared Function Max(ByVal val1 As String, ByVal val2 As String) As String
    Dim retVal As String = val1
    If (val2.CompareTo(val1) > 0) Then
        retVal = val2
    End If
    Return retVal
End Function
```

```
[C# code]
public static void Swap(ref String val1, ref String val2) {
    String tmpObj = val2;
    val2 = val1;
    val1 = tmpObj;
}

public static void Swap(ref Double val1, ref Double val2) {
    Double tmpObj = val2;
    val2 = val1;
    val1 = tmpObj;
}

public static int Max(int val1, int val2) {
    int retVal = val1;
    if (val2 > val1)
        retVal = val2;
    return retVal;
}

public static String Max(String val1, String val2) {
    String retVal = val1;
    if (val2.CompareTo(val1) > 0)
        retVal = val2;
    return retVal;
}
```

These examples include implementations of the `Swap()` and `Max()` methods. Methods of this nature are meant to invoke a general-purpose operation on an object without concern for its interface. `Swap()`, for example, simply causes two objects to trade places. `Max()` just determines and returns the maximum value of the two supplied parameters. However, in order to maintain type safety and avoid any boxing overhead for your value types, you are required to provide a series of overloaded versions of each of the methods.

In looking at these two methods, it's clear that expanding your list of overloads to embrace all types would be time consuming, would bloat your code, and would introduce maintenance overhead. However, using `object` types here would also be a mistake. It would introduce a host of other problems. It would also violate the spirit of Item 2.

This, of course, means the best option here is to make generic versions of these methods. The following represents generic implementations of the `Swap()` and `Max()` methods:

```
[VB code]
Public Shared Sub Swap(Of T)(ByRef val1 As T, ByRef val2 As T)
    Dim tmpObj As T = val2
    val2 = val1
```

```
        val1 = tmpObj
    End Sub

    Public Shared Function Max(Of T)(ByVal val1 As T, ByVal val2 As T) As T
        Dim retVal As T = val1
        If (Comparer(Of T).Default.Compare(val1, val2) < 0) Then
            retVal = val2
        End If
        Return retVal
    End Function
```

```
    [C# code]
    public static void Swap<T>(ref T val1, ref T val2) {
        T tmpObj = val2;
        val2 = val1;
        val1 = tmpObj;
    }

    public static T Max<T>(T val1, T val2) {
        T retVal = val1;
        if (Comparer<T>.Default.Compare(val1, val2) < 0)
            retVal = val2;
        return retVal;
    }
```

There's nothing earth-shattering about how generics make these methods better. These examples are only intended to represent a sample of a pattern you're going to want to look for in your own code. Essentially, anytime you find yourself overloading a method's signature to support variations of parameter types, you have to ask yourself if that method might be better implemented as a generic method. You'll also want to look at the body of these methods to determine how tightly they are coupled to the types that appear in their parameter lists.

As part of considering whether to make methods of this nature generic, you should also consider how constraints might be used to expose some minimal interfaces of your type parameters. If, for example, you were to constrain a method using `IComparable<T>`, you would be allowing the method to access the comparable interface without significantly narrowing the capabilities of the method. If your objects implement many of these general-purpose interfaces, these interfaces can then be leveraged as constraints and further expose the capabilities of your type parameters.

The thrust here, though, is to focus your energy on making cleaner, more type-safe replacements of existing methods. Any time you can reduce the size of code and simultaneously improve its type-safety, you need to seize the opportunity.

Replacing Multiple Delegates with One Generic Delegate

The introduction of generic delegates should fundamentally change how and when you create your own delegates. Delegates represent one of the most fundamental and natural applications of generics. As such, I have trouble imagining any situation where you would ever want to use a non-generic delegate. It is also possible that you may already have delegates in your code that could be improved via generics. Suppose, for example, you had the following non-generic delegates in your application:

```
[VB code]
Public Delegate Sub MyDel1(ByVal x As Int32, ByVal y As String)
Public Delegate Sub MyDel2(ByVal x As Int32, ByVal y As Double)
Public Delegate Sub MyDel3(ByVal x As Int32, ByVal y As Long)

Public Delegate Sub MyDel4(ByVal x As Int32, ByVal y As String, ByVal z As Double)
Public Delegate Sub MyDel5(ByVal x As Int32, ByVal y As Double, ByVal z As Double)
```

```
[C# code]
public delegate void MyDel1(int x, string y);
public delegate void MyDel2(int x, double y);
public delegate void MyDel3(int x, long y);

public delegate void MyDel4(int x, string y, double z);
public delegate void MyDel5(int x, double y, double z);
```

Here you have two sets of delegate signatures. The first set accepts two parameters and varies only by the type of the second parameter. The second set has three parameters and also varies only by its second parameter. Now, with these delegates in place, you can start declaring methods that implement these delegates. The question is, do you really need all of these declarations? No. You can actually replace all of these declarations with the following pair of generic delegates:

```
[VB code]
Public Delegate Sub MyDel(Of T, U)(ByVal x As T, ByVal y As U)
Public Delegate Sub MyDel(Of T, U, V)(ByVal x As T, ByVal y As U, ByVal z As V)
```

```
[C# code]
public delegate void MyDel<T, U>(T x, U y);
public delegate void MyDel<T, U, V>(T x, U y, V z);
```

These two generic delegate declarations will accept any permutation of types for your two- and three-parameter delegates, eliminating the need to declare a new delegate for each new method signature. This also improves the expressive qualities of those methods that accept delegates. The following code provides a few simple examples of how these generic delegates impact your interactions with methods that accept delegates:

```
[VB code]
Public Sub Func1(ByVal x As Int32, ByVal y As String)
End Sub

Public Sub Func4(ByVal x As Int32, ByVal y As String, ByVal z As Double)
End Sub

Public Sub AcceptDelegate(ByVal MyDel As MyDel(Of Int32, String))
End Sub

Public Sub AcceptDelegate(ByVal MyDel As MyDel(Of Int32, String, Double))
End Sub

Public Sub CallWithDelegate()
    AcceptDelegate(AddressOf Func1)
    AcceptDelegate(AddressOf Func4)
End Sub
```

```
[C# code]
public void Func1(int x, string y) {}
public void Func4(int x, string y, double z) { }

public void AcceptDelegate(MyDel<int, string> MyDel) { }
public void AcceptDelegate(MyDel<int, string, double> MyDel) { }

public void CallWithDelegate() {
    this.AcceptDelegate(Func1);
    this.AcceptDelegate(Func4);
}
```

Here, in your `AcceptDelegate()` method, you can see how your generic delegate is used to express the signature of methods that it will accept. This, from my perspective, clearly identifies the kinds of methods that can be supplied and makes it easier for you to change delegate method signatures directly at the spot where they are being used. In a non-generic model, you'd have to hunt down the delegate signature elsewhere to modify it.

The main idea here is that, for your existing code, you may have delegates that can be removed and replaced with generic delegates. There are certainly upsides here — especially in scenarios where you're looking for an approach that allows you to more clearly convey the signature of a delegate at the point where it is referenced.

Using Generic Methods as Delegate Methods

The methods that you supply as the implementation of a delegate may also be generic. These two constructs — used in combination — offer you a number of opportunities to reshape your approach to how you define and implement delegates in your solutions. At a minimum, as you look at each method that implements a given delegate, you should also consider whether a collection of delegate methods could be replaced by a single generic method. If this is the case, this would represent yet another opportunity to use generics to reduce the size and improve the maintainability of your delegate methods. It also puts you in a position where your delegate is prepared to support a broader set of types without any additional enhancement. Less can certainly translate into more in this scenario.

Introducing Generic Interfaces

Because interfaces don't contain implementation, they can represent a very natural target for applying generic concepts. The basic idea here, as it has been throughout this section, is to make more abstract representations of your interfaces that allows them to be applied to a broader set of data types. This is especially useful with interfaces that are more general in nature. In your own code, you should be looking for any interfaces that might benefit from the application of generics. Any interface that varies only by the type it leverages may be a candidate for generic refactoring.

The `IComparable<T>` and `IEnumerable<T>` interfaces make great examples of small, focused interfaces that leverage generics while remaining globally applicable to a wide variety of types. This characteristic also makes these same interfaces excellent candidates for being applied as constraints.

Balancing Readability with Expressiveness

Some view the syntactic constructs introduced by generics as a welcome addition to the language. This crowd looks at the type arguments that accompany a generic declaration and sees them as providing a

very precise, undeniably clear definition of each data type. For them, generics eliminate any confusion that might have been associated with using non-generic APIs.

Meanwhile, another population views generics as imposing on the readability of their code. They see type parameters and constraints and new keywords as muddying the image of what was an otherwise perfectly clean, uncluttered block of code. Many in this group see generics as undermining the general usability and maintainability of their code.

The challenge here is striking a balance between these two groups. If everyone can agree on the general value of generics, the only issue that remains is how to introduce them without creating code that is so confusing that it requires a decoder ring to decipher the text. The goal of this section is to offer up some guidelines that can establish some fundamental boundaries that give developers the freedom to leverage generics without leaving behind a trail of unreadable code.

Item 6: Use Expressive, Consistent Type Parameter Names

During the beta cycle for Visual Studio 2005, early adopters focused a significant portion of venom and debate on naming conventions for type parameters. Because type parameters are littered throughout your generic types, it makes sense that developers would be concerned about how these type parameters could be named in a manner that could accurately convey their intended use.

There are basically two camps of thought on this subject. One camp prefers single-letter type parameter names simply because they reduce the overall size of the signature of your generic declarations. This is a model that is employed by most C++ template libraries, which may contribute to the mindset of those who prefer to continue this tradition. The other camp finds these one-letter type parameters simply too terse. They don't see how a single letter can really adequately convey the nature of a type parameter. This group, as you might suspect, prefers lengthier, more expressive naming conventions. The following declarations illustrate the tradeoffs that are associated with these competing mindsets:

```
[VB code]
Public Class Dictionary(Of K, V)
End Class

Public Class Dictionary(Of TKey, TValue)
End Class

[C# code]
public class Dictionary<K, V> {}
public class Dictionary<TKey, TValue>
```

The first declaration is short and sweet, but hardly expressive. The second option uses full names, prepending a `T` to each name to designate it as a type parameter. In this scenario, the second of these two options seems like it might be the preferred model. However, consider this same approach as it might be applied to a generic method or delegate declaration with constraints applied. An example of this nature might appear as follows:

```
[VB code]
```

```
Public Function Foo(Of TKey As IComparable, TValue)(ByVal key As TKey, _
                                                    ByVal val As TValue) As TValue
End Function

Public Function Foo(Of K As IComparable, V)(ByVal key As K, ByVal val As V) As V
End Function
```

```
[C# code]
public TValue Foo<TKey, TValue>(TKey key, TValue val) where TKey : IComparable {}
public TValue Foo<K, V>(K key, V val) where K : IComparable {}
```

In this example, the longer names can get slightly more unwieldy. Naturally, the opposition would say the second of these two starts to resemble more of an algebraic equation than a method. At the same time, the full names certainly make this feel more like the signatures conform to a pattern that you might be more comfortable digesting.

The Naming Compromise

As you can imagine, there's no one guideline I can suggest that will suddenly resolve the preferences of either of these approaches. In the end, it's mostly subjective. Do you like spaces or tabs in your files? Do you indent your code two spaces or four? It almost falls into that area of debate that really ends up being more a matter of personal preference. Still, there are some guidelines in this area that should, at a minimum, establish some parameters for how you might standardize your approach.

The best compromise appears to be to use single-letter type parameters when a single letter adequately captures the nature of your type parameter. If, for example, you have a generic collection, the name `MyCollection<T>` would be considered acceptable. The use of `T` in this scenario is adequate, because a longer name can't really convey anything extra about the type parameter's intent or role. The truth is, in any scenario where you have a single un-constrained type parameter, the single-letter use of `T` will likely suffice.

However, cases exist where you have multiple type parameters playing specific, identifiable roles. In these situations, you should select longer, more expressive names that clearly convey the role of each type parameter. With a generic dictionary class, for example, you know its first type parameter represents a key and its second parameter represents a value. Given these roles, the guidelines suggest that you should declare this dictionary as `Dictionary<TKey, TValue>`. Here, you've added meaning to the names and conformed to the standard of pre-pending a `T` to each type parameter name.

Using Constraints to Qualify Names

If you're using constraints with your generic types, those constraints provide more information about the nature of the type parameters they constrain. Suppose, for example, you have the following declaration:

```
[VB code]
Public Class TestClass(Of T As IValidator)
    Private _myType As T
End Class
```

```
[C# code]
public class TestClass<T> where T : IValidator {
    private T _myType;
}
```

This example has a single type parameter that is constrained as being of the type `IValidator`. Now, as you reference the type parameter in the body of your class, the references to the type parameter as `T` does little to convey the fact that `T` is being used as an `IValidator` type. To remedy this, you should make the constraint name part of the type parameter name. The new, improved version of this declaration would appear as follows:

```
[VB code]
Public Class TestClass(Of TValidator As IValidator)
    Private _myType As TValidator
End Class

[C# code]
public class TestClass<T> where TValidator : IValidator {
    private TValidator _myType;
}
```

Now, as you reference your type parameter in the body of your class, the type parameter name provides significantly more insight into its nature. This will work for a number of scenarios. However, if you're using multiple constraints, you may opt to stick with a simple `T` as your type parameter name.

Generic Methods in Generic Classes

One area that seems to get left out of the naming debate is the name of type parameters for generic methods that appear within a generic class. Consider the following simple example:

```
[VB code]
Public Class TestClass(Of T)
    Public Sub Foo(Of T)(ByVal val As T)
        Dim localVar As T
    End Sub
End Class

[C# code]
public class TestClass<T> {
    public void Foo<T>() {
        T localVar;
    }
}
```

In conforming to the "use `T` when you have a single parameter" guideline, you have used `T` as the name of the type parameter for your class *and* you've used `T` as the type parameter for the method that appears within your class. Although this compiles, using the same type parameter name for both your class and the `Foo()` method creates a situation where your method will not be able to access the type parameter from its surrounding class.

This calls for another naming guideline that requires method type parameters to always be named in a consistent manner that will prevent methods from hiding access to the type parameters of their surrounding class. In cases where your method accepts a single parameter, I suggest that you use a consistent replacement for `T` that will be used across all your generic methods.

Being Consistent

Though I can espouse the value of naming conventions, I often doubt whether the masses can be persuaded to adopt a universal approach. That said, I do think there's room for agreement on the topic of consistency. No matter what scheme you adopt, it's essential that you be consistent with that theme

throughout your code. Consistency will contribute as much to the readability of your generic types as any guideline. Of course, if you're exposing your generic types as part of a public API, your choice of naming convention schemes gets more complicated. Ideally, you'd like your generic signatures and documentation to conform to a broader standard. This will simplify matters for the consumers of your API. So, in that light, you should place added value on making sure your naming conventions are keeping up with what, at this stage, is likely to continue to be a bit of a moving target.

Item 7: Use Aliasing for Complex or Frequently Used Types

In some cases, you may have a rather lengthy generic type declaration that is used heavily throughout a block of code. In these situations, you may find the bulky nature of your generic type as imposing on the overall readability of the rest of your code. Consider, for example, the following code:

```
[VB code]
Public Sub ProcessItem(ByVal value As MyType1(Of Long, Double, String), _
                                                        ByVal status As Int32)
    Dim x As New MyType1(Of Long, Double, String)
    If (status = 1) Then
        Dim y As MyType1(Of Long, Double, String) = value
    Else
        Dim z As New Nullable(Of MyType1(Of Long, Double, String))
    End If
End Sub
```

```
[C# code]
public void ProcessItem(MyType1<long, double, string> value, int status) {
   MyType1<long, double, string> x = new MyType1<long, double, string>();
   if (status == 1) {
       MyType1<long, double, string> y = value;
   } else {
       Nullable<MyType1<long, double, string>> z =
                                  new Nullable<MyType1<long, double, string>>();
   }
}
```

This method continually references the generic type `MyType1` and, with its three type arguments, it starts polluting the esthetics of your code in a hurry. Fortunately, for scenarios like this, you have the option of creating an alias that can act as a placeholder for these more heavyweight declarations. This detracts some from the expressiveness of your types but is worth it if the readability of your code is being compromised. Here's a quick look at how the aliased version of this function would improve the situation:

```
[VB code]
Imports MType = MyType1(Of Long, Double, String)

Public Class Aliasing
    Public Sub ProcessItem(ByVal value As MType, ByVal status As Int32)
        Dim x As New MType
        If (status = 1) Then
            Dim y As MType = value
        Else
            Dim z As New Nullable(Of MType)
        End If
    End Sub
End Sub
```

```
[C# code]
using MType = MyType1<long, double, string>;

public void ProcessItem(MType value, int status) {
    MType x = new MType();
    if (status == 1) {
        MType y = value;
    } else {
        Nullable<MType> z = new Nullable<MType>();
    }
}
```

An aliasing statement is added to the top of this example that now declares `MType`, which serves as a placeholder for the full generic declaration of `MyType1` throughout the implementation of your code. The result is certainly a more readable version of your method.

Item 8: Don't Use Constructed Types as Type Arguments

Although you may be embracing the splendor of generics, you still need to make sure you're not going overboard with your generic types. You can, if you choose, introduce generic types that can make construction a less than graceful process. Consider, for example, a generic type that accepts two constructed types as parameters:

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

Public Class MyType2(Of T, U, V)
End Class

Public Class MyType3(Of T, U)
End Class

Public Class TestClass
    Public Sub foo()
        Dim x As New MyComplexType(Of MyType2(Of Int32, String, Double), MyType3(Of
String, String))
    End Sub
End Class
```

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

public class MyType2<T, U, V> { }

public class MyType3<T, U> { }

public class TestClass {
    public void foo() {
      MyComplexType<MyType2<int, string, double>, MyType3<string, string>> x =
        new MyComplexType<MyType2<int, string, double>, MyType3<string, string>>();

    }
}
```

This example declares a generic class, `MyComplexClass`, which takes two type parameters. It also creates a couple of additional generic types that are then used as type arguments in the construction of an instance of `MyComplexClass`. You can see, from looking at this, that using constructed types as type arguments has a serious impact on the readability of your code. As a rule of thumb, you should avoid scenarios of this nature. This is not to say that you should completely abandon passing constructed types as parameters. It just means you should construct some intermediate representation and pass that declared type as your parameter. It's all about making your code more readable — not limiting what types can be used as parameters.

Item 9: Don't Use Too Many Type Parameters

The more type parameters you add to your generic types, the more difficult they will be to use and maintain. The reality is that there probably aren't too many situations where you will need to leverage more than two type parameters. This fact and the reality that using more than two type parameters is likely to negatively impact the usability of your generic types would suggest that, as a guideline, it would make sense to generally constrain the number of type parameters you use to two or less.

Item 10: Prefer Type Inference with Generic Methods

One of the best features of generic methods is their ability to infer the types of their type parameters. This feature eliminates the need to explicitly provide type arguments for each call to a generic method and, as a result, has a significant impact on the overall maintainability and readability of your code. The following example illustrates type inference in action:

```
[VB code]
Public Class TypeInference
    Public Sub MyInferenceMethod(Of I, J)(ByVal param1 As I, ByVal param2 As J)
    End Sub

    Public Sub MakeInferenceCall ()
        MyInferenceMethod("TestVal", 122)
        MyInferenceMethod(122, "TestVal")
        MyInferenceMethod(New Order(), 833.22)
    End Sub
End Class
```

```
[C# code]
public class TypeInference {
    public void MyInferenceMethod<I, J>(I param1, J param2) { }

    public void MakeInferenceCall () {
        MyInferenceMethod("TestVal", 122);
        MyInferenceMethod(122, "TestVal");
        MyInferenceMethod(new Order(), 833.22);
    }
}
```

From this example you can see how type inference makes the generic-ness of your method completely transparent. Each call that is made here is no different than the calls you might make to a non-generic method. It's as if you've overloaded this method with every combination of possible data types. Given this upside, it's only natural to have a guideline that suggests that consumers of generic methods should, as a rule of thumb, always prefer type inference to explicitly specified type arguments.

There are also times when the declaration of your generic methods will prohibit you from inferring the types of your parameters. Consider the following:

```
[VB code]
Public Function MyNonInferenceMethod(Of I, J)() As I
End Function

[C# code]
public I MyNonInferenceMethod<I, J>() { }
```

The absence of references to type parameters in the signature of this method means there's no way to supply types that could then be used to infer type information for your type parameters. This isn't necessarily wrong or even common. However, you should still be aware of the fact that clients of this method will not be able to leverage type inference when making their calls.

Item 11: Don't Mix Generic and Non-Generic Static Methods

If you are using static methods in a generic class that also includes static *generic* methods, you may end up creating some ambiguity in the interface of your class. The following example illustrates a simple case where this could be a problem:

```
[VB code]
Public Class TestClass(Of T)
    Public Shared Sub Foo()
    End Sub

    Public Shared Sub Foo(Of T)()
    End Sub
End Class

[C# code]
public class TestClass<T> {
    public static void Foo() {}
    public static void Foo<T>() {}
} Okay - MAS
```

You'll notice that this generic class includes a static method `Foo()` in addition to a static generic method also named `Foo()`. Given these two methods, imagine the confusion this would generate for consumers of this class. The following provides an example of calls to both of these methods:

```
[VB code]
TestClass(Of String).Foo()
TestClass.Foo(Of String)()

[C# code]
TestClass<string>.Foo();
TestClass.Foo<string>();
```

The presence of both of these static methods, as you can see, ends up creating some general confusion. Fortunately, you're not likely to end up in many scenarios like this one. Or, if you do, you could easily overcome this by altering your method names.

Using BCL Generic Types

The `System.Collections.Generic` and `System.Collections.ObjectModel` namespaces introduce a whole host of out-of-the-box generic types. Chapter 8, "BCL Generics," looks at each of these types in great detail. However, there are also some simple guidelines you need to keep in mind as you work with these types. The items that follow point out a few key areas you'll want to consider as part of working with this set of types.

Item 12: Custom Collections Should Extend Collection<T>

At some point you're likely to want to introduce your own generic custom collections. These collections are typically implemented as extensions to one of the existing collection classes. This allows them to inherit all the behavior of the existing collection and supplement or amend that functionality with new operations that are targeted at addressing the specific requirements of your solution.

In these situations, you may be tempted to have your custom collections be implemented as an extension of the `List<T>` class. `List<T>` is certainly the most robust and powerful of the containers found in the `System.Collections.Generic` namespace. However, to achieve its optimizations, this class also prevents clients from overriding or altering most of its behavior. Suppose, for example, you wanted to amend the list class and record some additional data each time an item was added to or removed from a list. With the `List<T>` class, you would not be allowed to override the methods that clients use to add and remove items.

So, although `List<T>` may be one of your favorite classes to consume, it is not intended to serve as the base class for your custom collections. Instead, the `Collection<T>` class is meant to play this role. Although it doesn't have all the capabilities of `List<T>`, it exposes a key set of protected members that you can freely override in your descendent types.

The `System.Collections.ObjectModel` namespace provides two additional collection implementations that should be lumped into this category with the `Collection<T>` class. The `ReadOnlyCollection<T>` and `KeyedCollection<TKey, TItem>` classes provide specific variations of the `Collection<T>` class and both are intentionally open to further specialization. The theme here is that this family of `xxxxCollection` classes are all meant to serve as the common foundation for any custom collection classes you might want to introduce. This, I believe, is part of the rationale behind having these types appear in their own, separate namespace.

As you create specializations of these collection classes, you should attempt to apply a naming scheme that allows each new collection class to continue to convey the nature of the class. A descendant of `KeyCollection<TKey, TItem>`, for example, might be called `MyKeyedCollection<TKey, TItem>`. This makes the role of the collection very explicit to consumers of that collection.

Item 13: Use the Least Specialized Interface in Your APIs

The collections included in the `System.Collections.Generic` namespace implement a series of different interfaces that provide varying levels of support for interacting with and managing your collections. When using these types in your own APIs, you should give special consideration to which interface best suits your requirements. As a rule of thumb, you should select the least specialized interface in these scenarios. If, for example, you're just going to iterate over the collection's items sequentially, you only need the `IEnumerable<T>` interface. However, if you want index-based access, you

may want to consider using `IList<T>`. If you'll be modifying the state of the collection, you may need to consider using `ICollection<T>`.

The basic idea here is that, by choosing the least specialized interfaces, you're able to limit the constraints placed on the clients of your types. You should always factor this into your thinking when selecting a generic interface for inclusion in your own APIs. This guideline is really a more general, OOP guideline. However, it seems worth highlighting again in the context of generics.

Item 14: Enable "for each" Iteration with IEnumerable<T>

The `System.Collections.Generic` namespace includes an `IEnumerable<T>` interface. This interface provides a standard mechanism for iterating over the items in a collection. However, its role is more significant than the other collection-based interfaces that are part of the framework. This interface is what enables, indirectly, the mechanism that is employed by the `foreach` construct.

In general, the `foreach` construct is often viewed as the preferred mechanism for sequentially processing the items in a collection, providing a cleaner, more readable approach to processing the items in a collection. Given these realities, it's fair to assume that consumers of your custom collections are going to expect you to provide support for `foreach`-based iteration. So, as a general rule of thumb, you should always consider implementing the `IEnumerable<T>` interface on any custom type that needs to support iteration.

Applying Constraints

Anytime you choose to apply constraints to your generic types, you're narrowing the applicability and reusability of that type. And, via constraints, you're also influencing heavily the scope of what can be achieved within the implementation of your generic types. The next set of items point out some specific topics you'll want to consider as you apply constraints to your type parameters.

Item 15: Select the Least Restrictive Constraints

In selecting an appropriate constraint for a type parameter, you should attempt to choose the constraint that gives you the minimum level of accessibility you need without imposing any unneeded, additional constraints on your type parameters. Here's a quick example that illustrates how a constraint might be overly restrictive:

```
[VB code]
Public Interface IPerson
    Sub Validate()
End Interface

Public Interface ICustomer
    Inherits IPerson
End Interface

Public Interface IEmployee
    Inherits IPerson
End Interface
```

```
Public Class TestClass(Of T As ICustomer)
    Public Sub New(ByVal val As T)
        val.Validate()
    End Sub
End Class
```

```
[C# code]
public interface IPerson {
    void Validate();
}

public interface ICustomer : IPerson { }

public interface IEmployee : IPerson { }

public class TestClass<T> where T : ICustomer {
    public TestClass(T val) {
        val.Validate();
    }
}
```

In this example, you have a hierarchy of interfaces where `IPerson` is at the base and has descendent interfaces for `ICustomer` and `IEmployee`. Now, in your `TestClass`, you're currently expecting it to primarily work with `Customer` types and, because you need to call `Validate()` on each customer in the constructor, you've applied the `ICustomer` constraint to your type parameter to enable access to this method.

This certainly works. However, it also overly constrains your type parameter. Because all you're accessing at this point is the `Validate()` method and that method is part of the `IPerson` interface, you should have used `IPerson` to constrain your type parameter. That opens up your type to support other types that implement `IPerson`. The rule of thumb here is that you want to select the least constraining interface that still allows you to satisfy the compile-time validation of your type. If you need to be more restrictive, you can always alter the constraints as your solution evolves.

Item 16: Don't Impose Hidden Constraints

Even though you may have a type that does not include any constraints in its declaration, that does not mean that your type can't impose "hidden" constraints within its implementation. You can imagine how, through a cast or through calls to the `GetType()` method, you could create code within your generic type that builds in some assumptions about the nature of its type parameters. In these situations, you are still imposing, indirectly, constraints on your type parameter — they're just not being explicitly declared. In the end, these are still viewed as constraints and they're still considered a bad idea.

Item 17: Avoid Multiple Constraint Ambiguity

When you're working with constraints, you have the option of applying multiple constraints to any type parameter. In fact, you can combine a single class constraint with multiple interface constraints. As you start to mix and match multiple constraints, you can end up introducing ambiguity within the scope of your generic type. Consider the following example:

```
[VB code]
Public Interface I
    Sub Foo1()
    Sub Foo3()
End Interface

Public Class C
    Public Sub Foo1()
    End Sub
End Class

Public Class TestClass(Of T As C, I)
End Class
```

```
[C# code]
public interface I {
    void Foo1();
    void Foo3();
}

public class C {
    public void Foo1() {}
}

public class TestClass<T> where T : C, I { }
```

This example declares a class that employs class and interface constraints, both of which share a common method, `Foo1()`. Though rules exist that will essentially force the class constraints to take precedence over the interface constraints here, this situation is still ambiguous at best. From my perspective, this represents a scenario you should attempt to avoid. Fortunately, applying constraints that are likely to overlap in this manner should be rare.

Item 18: Provide Parameterless Constructors

Whenever you're introducing your own types, you want to consider how those types will behave when used as a type argument. Obviously, the interfaces you choose to implement will play a key role in how that type can be constrained. At a minimum, every type you want to use as a type argument should include support for parameterless construction. By supporting this constraint, you enable your type to be supplied as a type argument to any generic type that includes a constructor constraint.

Plenty of instances exist where supporting parameterless construction adds value to your interface — for generic and non-generic solutions. If you've worked at all with any variant of the factory pattern, you've probably already provided a parameterless constructor. And, with generics, the list of scenarios where this adds value just gets longer. Suppose, for example, you had the following generic method that retrieved a collection of items from the database:

```
[VB code]
Public Function GetDataObjects(Of T As New)() As IEnumerable(Of T)
End Function
```

```
[C# code]
public IEnumerable<T> GetDataObjects() where T : new() {}
```

This method looks up data objects and returns them in a generic collection. It leverages the incoming type parameter to populate this collection with specific types. However, it would not be able to achieve this without being able to apply the constructor constraint to your incoming type parameter. So, any type you want to use with this method must support a parameterless constructor.

The Kitchen Sink

In addition to the guidelines discussed previously, a handful of items also exist that don't necessarily fit into any specific categories. The items that appear in the sections that follow fall into this grab bag of miscellaneous items.

Item 19: Use Static Data Members with Caution

Outside of generics, the behavior of static data members is well understood. Basically, when a data member is static, this indicates that there is one and only one instance of that data member for *all* instances of that class. This is where the VB "shared" keyword almost conveys the concept better than "static" in that these members are actually shared by all instances of the class.

With generics, the behavior of static data members may not actually match what you're expecting. Take a look at a small example that illustrates how generics manage static data:

```
[VB code]
Public Class StaticData(Of T)
    Private Shared _staticData As Int32 = 0

    Public Sub IncrementCount()
        _staticData = _staticData + 1
    End Sub
End Class

Public Sub TestStaticData()
    Dim instance1 As New StaticData(Of String)()
    instance1.IncrementCount()

    Dim instance2 As New StaticData(Of Int32)()
    instance2.IncrementCount()

    Dim instance3 As New StaticData(Of String)()
    instance3.IncrementCount()
End Sub
```

```
[C# code]
public class StaticData<T> {
    private static int _staticData = 0;

    public void IncrementCount() {
        _staticData++;
    }
}
```

```
public void TestStaticData() {
    StaticData<String> instance1 = new StaticData<String>();
    instance1.IncrementCount();

    StaticData<int> instance2 = new StaticData<int>();
    instance2.IncrementCount();

    StaticData<String> instance3 = new StaticData<String>();
    instance3.IncrementCount();
}
```

This example uses a simple generic class that has a static data member. In the `TestStaticData()` method, you declare three separate instances of this class and increment the count held by the static data member. Now, for a non-generic class, the static data member would end up being shared among all instances of the `StaticData` class. At the end of executing this code, the static data member would have been incremented to a value of 3.

With generic types, though, static data members are static for all constructed types that have the same type arguments. Looking back at the example, you'll notice that `instance1` uses a `string` type argument and `instance2` supplies an `integer` type argument. So, these two instances each have their own static data member and, therefore, their values are also incremented separately.

The last instance declared here, `instance3`, uses the same type argument as `instance1`. In this case, because the type arguments of these two instances match, they will end up sharing a common static data member. This means the increment performed on `instance3` will bump the count up to 2.

Once you understand what's going on here, it makes sense. At the same time, if you just look at the code, this side effect may not always be anticipated. And, if you distribute these calls out across a larger body of more complex code, you can imagine scenarios in which this could introduce some difficult-to-detect bugs. I'm not sure if it's completely accurate to classify this as a guideline. However, in the spirit of "good practices," you're likely to want to be especially careful about how you use static data members with generic types.

Item 20: Use Interfaces in Lieu of Classes

The non-generic APIs you've constructed and consumed have probably relied heavily on the use of interfaces. It's a common practice. An interface allows you to express the signature of a type without binding to any specific implementation of that type. As such, an interface is especially useful in terms of how it shapes your API, enabling its types to be expressed in the manner that allows a single signature to accept multiple implementations.

This pre-existing idea of preferring interfaces to classes in your APIs carries forward into the world of generics. Consider, for example, the following scenario:

```
[VB code]
Public Class DataAccessMgr(Of T, U)
    Public Function FindItems(ByVal id As Int32) As List(Of T)
    End Function

    Public Sub AddChildren(ByVal parent As T, ByVal dataObjects As List(Of U))
    End Function
End Class
```

```
[C# code]
public class DataAccessMgr<T, U> {
    public List<T> FindItems(int id) {}
    public void AddChildren(T parent, List<U> dataObjects) {}
}
```

There's nothing outright wrong with the methods in this class. However, you'll notice that this class makes the mistake of referencing the `List<T>` in its interface. Using `List<T>` here forces all clients of this class to use `List<T>` classes in their interactions with the API.

If, within the body of the `AddChildren()` method, you're just iterating over the items in the list, you don't really need to be bound directly to the concrete `List<T>` class. Instead, you can change the signature of this method to use one of the generic collection interfaces, say `IEnumerable<T>`, and still have no impact on the internal implementation of this method. This allows your method to accept multiple implementations of `IEnumerable<T>` without restricting what can be achieved within the implementation of your method.

The real focus here is on providing API interfaces that allow type safety without somehow imposing artificial restrictions on your API. You should familiarize yourself with all the interfaces that are part of the `System.Collections.Generic` namespace so you'll have a better feel for which interfaces convey the meaning you want expressed by your API. You should also keep this general guideline in mind as you create your own generic interfaces.

Item 21: Use Comparer<T> for All Type Comparisons

`Comparer<T>` is one of those nice, type-safe utility classes that you almost get for free when you add generics to a language. `Comparer<T>` should be viewed as the replacement for all the previous modes of type comparison that have been historically used in the pre-generic era.

Item 22: Use Nullable<T> for Optional Values

When working with value types, there's no standard that defines a state that represents "empty" or "null." Each solution is left to its own devices for defining this state. As discussed before, `Nullable<T>` allows you to overcome this, providing you with a standardized mechanism for determining if a value type has actually been assigned a value.

As a guideline, it's fair to say that you should consider using `Nullable<T>` for any value type where you may need to determine its null state. A variety of common scenarios exist where this type can be applied. When working with databases, for example, you might want to use `Nullable<T>` to determine if a column is empty or null. Or, if you're working with a list of properties, you may apply `Nullable<T>` to accurately capture the state of a given property. These are just a couple of the more obvious applications of this type. I'm sure, from your own experiences, you can imagine a whole host of situations where you could see yourself leveraging this type.

Item 23: Use EventHandler<T> for All Events

Item 5 talked about the general value of using a single generic delegate as a replacement for many separate, non-generic delegate declarations. And, because delegates are used as part of eventing, this same idea ends up generating a related guideline around the use of the `EventHandler<T>` type. This type

will allow you to declare event handler delegates without having to create one-off declarations for each delegate. This solution is much cleaner than using non-generic event handlers and should be used as the default for all event handling declarations.

Summary

The goal of this chapter was to introduce a series of guidelines that could help provide some general rules that would help refine your approach to consuming or constructing generic types. This chapter looked at some specific patterns where generics might be applied as alternatives to existing, non-generic solutions. It also looked at guidelines in the areas of usability, applying constraints, and using BCL generic types. It's important to note that these guidelines are just that — guidelines. They are not accepted standards or hard-and-fast rules. Instead, they represent suggestions for common themes you want to consider as you begin to exercise generic types. As generics mature, however, you can expect these guidelines to evolve, grow, and ultimately become more widely accepted as industry standards.