Generics FAQ

By Juval Lowy

MSDN, October 2005

Generics FAQ

Fundamentals

What is a generic type?

A *generic type* is a type that uses generic type parameters. For example, the type `LinkedList<K,T>`, defined as:

[C#]

```
public class LinkedList<K,T>
{...}
```

[VB]

```
Public Class LinkedList(Of K, T)
    ...
End Class
```

[C++]

```
generic <typename K, typename T>
public ref class LinkedList
{...};
```

is a generic type, because it uses the generic type parameters `K` and `T`, where `K` is the list's key and `T` is the type of the data item stored in the list. What is special about generic types is that you code them once, yet you can use them with different parameters. Doing so has significant benefits – you reuse your development and testing efforts, without compromising type safety and performance, and without bloating your code.

What is a generic type parameter?

A *generic type parameter* is the place holder a generic type uses. For example, the generic type `LinkedList<K,T>`, defined as:

[C#]

```
public class LinkedList<K,T>
{...}
```

[VB]

```
Public Class LinkedList(Of K, T)
    ...
End Class
```

[C++]

```
generic <typename K, typename T>
public ref class LinkedList
{...};
```

uses two type parameters - `K` and `T`, where `K` is the list's key and `T` is the type of the data item stored in the list. Using generic type parameters allows the linked list to defer the decision on the actual types to use. In fact, it is up to the client of the generic linked list to specify the generic type parameters to use.

What is a generic type argument?

A *generic type argument* is the type the client specifies to use instead of the type parameter. For example, given this generic type definition and declaration:

[C#]

```
public class MyClass<T>
{…}
MyClass<int> obj = new  MyClass<int>();
```

[VB]

```
Public Class SomeClass(Of T)
    ...
End Class
Dim obj As New SomeClass(Of Integer)
```

[C++]

```
generic <typename T>
public ref class MyClass
{…};
MyClass<int> ^obj = gcnew  MyClass<int>;
```

`T` is the type parameter, while integer is the type argument.

What is a constructed type?

A *constructed type* is any generic type that has at least one type argument.

For example, given this generic linked list definition:

[C#]

```
public class LinkedList<T>
{...}
```

[VB]

```
Public Class LinkedList(Of T)
    ...
End Class
```

[C++]

```
generic <typename T>
public ref class LinkedList
{...};
```

Then the following is a constructed generic type:

[C#]

```
LinkedList<string>
```

[VB]

```
LinkedList(Of String)
```

[C++]

```
LinkedList<String ^>
```

To qualify as a constructed type you can also specify type parameters to the generic type:
[C#]

```
public class MyClass<T>
{
   LinkedList<T> m_List; //Constructed type
}
```

[VB]

```
Public Class SomeClass(Of T)
    Dim m_List As LinkedList(Of T) ' Constructed type
End Class
```

[C++]

```
generic <typename T>
public ref class MyClass
{
   LinkedList<T> ^m_List; //Constructed type
};
```

What is an open constructed type?

A *open constructed type* is any generic type that which contains at least one type parameter used as a type argument. For example, given this definition:

[C#]

```
public class LinkedList<K,T>
{...}
```

[VB]

```
Public Class LinkedList(Of K, T)
    ...
End Class
```

[C++]

```
generic <typename K, typename T>
public ref class LinkedList
{...};
```

Then the following declarations of `LinkedList<K,T>` member variables are all open constructed types:

[C#]

```
public class MyClass<K,T>
{
   LinkedList<K,T>      m_List1; //Open constructed type
   LinkedList<K,string> m_List2; //Open constructed type
   LinkedList<int,T>    m_List3; //Open constructed type
}
```

[VB]

```
Public Class SomeClass(Of K, T)
    Dim m_List1 As LinkedList(Of K, T)       'Open constructed type
    Dim m_List2 As LinkedList(Of K, String)  'Open constructed type
    Dim m_List3 As LinkedList(Of Integer, T)'Open constructed type
```

```
End Class
```

[C++]

```
generic <typename K, typename T>
public ref class MyClass
{
   LinkedList<K, T>      ^m_List1; //Open constructed type
   LinkedList<K, String ^> ^m_List2; //Open constructed type
   LinkedList<int, T>    ^m_List3; //Open constructed type
};
```

What is a closed constructed type?

A *closed constructed type* is a generic type that which contains no type parameters as type arguments. For example, given this definition:

[C#]

```
public class LinkedList<K,T>
{...}
```

[VB]

```
Public Class LinkedList(Of K, T)
    ...
End Class
```

[C++]

```
generic <typename K, typename T>
public ref class LinkedList
{...};
```

Then the following declarations of `LinkedList<K,T>` member variables are all closed constructed types:

[C#]

```
LinkedList<int,string> list1; //Closed constructed type
LinkedList<int,int>    list2; //Closed constructed type
```

[VB]

```
Dim list1 As LinkedList(Of Integer, String) 'Closed constructed type
Dim list2 As LinkedList(Of Integer, Integer)'Closed constructed type
```

[C++]

```
LinkedList<int, String ^> ^list1; //Closed constructed type
LinkedList<int, int>      ^list2; //Closed constructed type
```

How do I use a generic type?

or

How do I initialize a generic type parameter?

When declaring a generic type, you need to specify the types that will replace the type parameters in the declaration. These are known as type arguments to the generic type. Type arguments are simply types. For example, when using this generic linked list:

[C#]

```
public class LinkedList<K,T>
{
   public void AddHead(K key,T item);
   //Rest of the implementation
}
```

[VB]

```
Public Class LinkedList(Of K, T)
    Public Sub AddHead(ByVal key As K, ByVal item As T)
    'Rest of the implementation
End Class
```

[C++]

```
generic <typename K, typename T>
public ref class LinkedList
{
   public: void AddHead(K key,T item);
   //Rest of the implementation
};
```

You need to specify which types to use for `K`, the list's key, and `T`, the data items stored in the list. You specify the types in two places: when declaring the list's variable and when instantiating it:

[C#]

```
LinkedList<int,string> list = new LinkedList<int,string>();
list.AddHead(123,"ABC");
```

[VB]

```
Dim list As New LinkedList(Of Integer, String)
list.AddHead(123, "ABC")
```

[C++]

```
LinkedList<int, String ^> ^list = gcnew LinkedList<int, String ^>;
list->AddHead(123,"ABC");
```

Once you specify the types to use, you can simply call methods on the generic type, providing appropriate values of the previously specified types.

A generic type that has type arguments already, such as `LinkedList<int,string>` is called a constructed type.

When specifying type arguments for generic types, you can actually provide type parameters. For example, consider this definition of the `Node<K,T>` class, which is used as a node in a linked list:

[C#]

```
class Node<K,T>
{
   public K Key;
   public T Item;
   public Node<K,T> NextNode;

   public Node(K key,T item,Node<K,T> nextNode)
   {
```

```
      Key      = key;
      Item     = item;
      NextNode = nextNode;
   }
}
```

[VB]

```
Class Node(Of K, T)
    Public Key As K
    Public Item As T
    Public NextNode As Node(Of K, T)
    Public Sub Node(ByVal key As K, ByVal item As T, ByVal nextNode As Node(Of K,
T))
        Me.Key = key
        Me.Item = item
        Me.NextNode = nextNode
    End Sub
End Class
```

[C++]

```
generic <typename K, typename T>
ref class Node
{
public:
   K Key;
   T Item;
   Node<K,T> ^NextNode;

   Node(K key,T item,Node<K,T> ^nextNode)
   {
      Key      = key;
      Item     = item;
      NextNode = nextNode;
   }
};
```

The `Node<K,T>` class contains as a member variable a reference to the next node. That member must be provided with the type to use instead of its generic type parameters. The node specifies its own type parameters in this case.

Another example of specifying generic type parameters to a generic type is how the linked list itself may declare and use the node:

[C#]

```
public class LinkedList<K,T>
{
   Node<K,T> m_Head;

   public void AddHead(K key,T item)
   {...}
}
```

[VB]

```
Public Class LinkedList(Of K, T)
    Dim m_Head As Node(Of K, T)
    Public Sub AddHead(ByVal key As K, ByVal item As T)
```

```
        ...
    End Sub
End Class
```

[C++]

```
generic <typename K, typename T>
public ref class LinkedList
{
   Node<K,T> ^m_Head;

   public: void AddHead(K key,T item)
   {...}
};
```

Note that the use of `K` and `T` in the linked list as the names of the type arguments is purely for readability purposes, to make the use of the node more consistent. You could have defined the linked list with any other generic type parameter names, in which case, you need to pass them along to the node as well:

[C#]

```
public class LinkedList<Key,Item>
{
   Node<Key,Item> m_Head;

   public void AddHead(Key key,Item item)
   {...}
}
```

[VB]

```
Public Class LinkedList(Of Key, Item)
    Dim m_Head As Node(Of Key, Item)
    Public Sub AddHead(ByVal key As Key, ByVal item As Item)
        ...
    End Sub
End Class
```

[C++]

```
generic <typename Key, typename Item>
public ref class LinkedList
{
   Node<Key,Item> ^m_Head;
   public: void AddHead(Key key,Item item)
   {...}
};
```

What are the benefits of generics?

Without generics, if you would like to develop general-purpose data structures, collections or utility classes, you would have to base all those on `object`. For example, here is the `object`-based `IList` interface, found in the `System.Collections` namespace:

[C#]

```
public interface IList : ICollection
{
```

```
    int Add(object value);
    bool Contains(object value);
    int IndexOf(object value);
    void Insert(int index, object value);
    void Remove(object value);
    object this[int index]{ get; set; }
    //Additional members
}
```

[VB]

```
Public Interface IList
    Inherits ICollection

    Function add(ByVal value As Object) As Integer
    Function contains(ByVal value As Object) As Boolean
    Sub insert(ByVal index As Integer, ByVal value As Object)
    Sub Remove(ByVal value As Object)
    Property Item(ByVal index As Integer) As Object
    'Additional members
End Interface
```

[C++]

```
public interface class IList : ICollection, IEnumerable
{
    int Add(Object ^value);
    bool Contains(Object ^value);
    void Insert(int index, Object ^value);
    void Remove(Object ^value);
    property Object ^ default[]
     {
          Object ^ get(int index);
          void set(int index, Object ^value);
     }
    //Additional members
};
```

Clients of this interface can use it to manipulate linked lists of any type, including value types such as integers or reference types such as strings:

[C#]

```
IList numbers = new ArrayList();
numbers.Add(1); //Boxing
int number = (int)numbers[0];//Unboxing

IList names = new ArrayList();
names.Add("Bill");
string name = (string)names[0];//Casting
```

[VB]

```
Dim numbers As IList = New ArrayList()
numbers.Add(1) 'Boxing
Dim number As Integer = CType(numbers(0), Integer)'Unboxing
Dim names As IList = New ArrayList()
names.Add("Bill")
Dim name As String = CType(names(0), String)'Casting
```

[C++]

```
IList ^numbers = gcnew ArrayList;
numbers->Add(1); //Boxing
int number = (int)numbers[0];//Unboxing

IList ^names = gcnew ArrayList;
names->Add("Bill");
String ^name = (String ^)names[0];//Casting
```

However, because `IList` is `object`-based, every use of a value type would force boxing it in an object, and unboxing it when using the indexer. Use of reference types forces the use of a cast which both complicates the code and has an impact on performance.

Now, consider the generics-equivalent interface, `IList<T>`, found in the `System.Collections.Generic` namespace:

[C#]

```
public interface IList<T> : ICollection<T>
{
   int IndexOf(T item);
   void Insert(int index, T item);
   void RemoveAt(int index);
   T this[int index]{ get; set; }
}
```

[VB]

```
<DefaultMember("Item")> _
Public Interface IList(Of T)
    Inherits ICollection(Of T)

    Function IndexOf(ByVal item As T) As Integer
    Sub Insert(ByVal index As Integer, ByVal item As T)
    Sub RemoveAt(ByVal index As Integer)
    Property Item(ByVal index As Integer) As T
End Interface
```

[C++]

```
generic <typename T>
public interface class IList : ICollection<T>
{
   int IndexOf(T item);
   void Insert(int index, T item);
   void RemoveAt(int index);
   property T default[]
   {
      T get(int index);
      void set(int index, T value);
   }
   //Additional members
};
```

Clients of this `IList<T>` can also use it to manipulate linked lists of any type, but doing so without any performance penalties. When using a value type instead of the type

parameters, no boxing or unboxing is performed, and when using a reference type, no cast is required:

[C#]

```
IList<int> numbers = new List<int>();
numbers.Add(1);
int number = numbers[0];

IList<string> names = new List<string>();
names.Add("Bill");
string name = names[0];
```

[VB]

```
Dim numbers As IList(Of Integer) = New List(Of Integer)()
numbers.Add(1)
Dim number As Integer = numbers(0)

Dim names As IList(Of String) = New List(Of String)()
names.Add("Bill")
Dim name As String = names(0)
```

[C++]

```
IList<int> ^numbers = gcnew List<int>;
numbers->Add(1);
int number = numbers[0];

IList<String ^> ^names = gcnew List<String ^>;
names->Add("Bill");
String ^name = names[0];
```

Various benchmarks have shown that in intense calling patterns, generics yield on average 200% performance improvement when using value types, and some 100% performance improvement when using reference types.

However, performance is not the main benefit of generics. In most real-life applications, bottle necks such as I/O will mask out any performance benefit from generics. The most significant benefit of generics is type-safety. With the `object`-based solutions, mismatch in type will still get complied, but yield an error at runtime:

[C#]

```
IList numbers = new ArrayList();
numbers.Add(1);
string name = (string)numbers[0]; //Run-time error
```

[VB]

```
Public Class SomeClass
   ...
End Class

Dim numbers As IList = New ArrayList()
numbers.Add(1)
Dim obj As SomeClass = CType(numbers(0), SomeClass) 'Run-time error
```

[C++]

```
IList ^numbers = gcnew ArrayList;
numbers->Add(1);
String ^name = (String ^)numbers[0]; //Run-time error
```

In large code bases, such errors are notoriously difficult to track down and resolve. With generics, such code would never get compiled:

[C#]

```
IList<int> numbers = new List<int>();
numbers.Add(1);
string name = numbers[0]; //Compile-time error
```

[VB]

```
Public Class SomeClass
   ...
End Class

Dim numbers As IList(Of Integer) = New List(Of Integer)()
numbers.Add(1)
Dim obj As SomeClass = numbers(0)'Compile-time error
```

[C++]

```
IList<int> ^numbers = gcnew List<int>;
numbers->Add(1);
String ^name = numbers[0]; //Compile-time error
```

Why can't I use type-specific data structures instead of generics?

To avoid the type-safety problem without generics, you might be tempted to use type-specific interfaces and data structure, for example:

[C#]

```
public interface IIntegerList
{
   int Add(int value);
   bool Contains(int value);
   int IndexOf(int value);
   void Insert(int index, int value);
   void Remove(int value);
   int this[int index]{ get; set; }
   //Additional members
}
```

[VB]

```
Public Interface IIntegerList
    Function Add(ByVal value As Integer) As Integer
    Function Contains(ByVal value As Integer) As Boolean
    Function IndexOf(ByVal value As Integer) As Integer
    Sub Insert(ByVal index As Integer, ByVal value As Integer)
    Sub Remove(ByVal value As Integer)
    Property Item(ByVal index As Integer) As Integer
    'Additional members
End Interface
```

[C++]

```
public interface class IIntegerList
{
   int Add(int value);
   bool Contains(int value);
   int IndexOf(int value);
   void Insert(int index, int value);
   property int default[]
    {
         int get(int index);
         void set(int index, int value);
    }
   //Additional members
};
```

The problem with that approach is that you will need a type-specific interface and implementation per data type you need to interact with, such as a `string` or a `Customer`. If you have a defect in your handling of the data items, you will need to fix it in as many places as types, and that is simply error-prone and impractical. With generics, you get to define and implement your logic once, yet use it with any type you want.

When should I use generics?

You should use generics whenever you have the option to. Meaning, if a data structure or a utility class offers a generic version, you should use the generic version, not the `object`-based methods. The reason is that generics offer significant benefits, including productivity, type safety and performance, at literally no cost to you. Typically, collections and data structures such as linked lists, queues, binary trees etc will offer generics support, but generics are not limited to data structures. Often, utility classes such as class factories or formatters also take advantage of generics. The one case where you should not take advantage of generics is cross-targeting. If you develop your code to target .NET 1.1 or earlier, then you should not use any of the new .NET 2.0 features, including generics. In C# 2.0, you can even instruct the compiler in the project settings (under Build | Advanced) to use only C# 1.0 syntax (ISO-1).

Are generics covariant, contra-variant or invariant?

Generic types are not covariant. Meaning, you cannot substitute a generic type with a specific type argument, with another generic type that uses a type argument that is the base type for the first type argument. For example, the following statement does not compile:

[C#]

```
class MyBaseClass
{}
class MySubClass : MyBaseClass
{}
class MyClass<T>
{}
//Will not compile
MyClass<MyBaseClass> obj = new MyClass<MySubClass>();
```

[VB]

```
Public Class MyBaseClass
    ...
End Class
Public Class MySubClass
    Inherits MyBaseClass
    ...
End Class
Public Class SomeClass(Of T)
    ...
End Class
'Will not compile.
Dim obj As SomeClass(Of MyBaseClass) = New SomeClass(Of MySubClass)()
```

[C++]

```
ref class MyBaseClass
{};
ref class MySubClass : MyBaseClass
{};
generic <typename T> where T : MyBaseClass
ref class MyClass
{};
//Will not compile
MyClass<MySubClass ^> ^obj = gcnew MyClass<MyBaseClass ^>;
```

[C#]

Using the same definition as in the example above, it is also true that `MyClass<MyBaseClass>` is not the base type of `MyClass<MySubClass>`:

```
Debug.Assert(typeof(MyClass<MyBaseClass>) != typeof(MyClass<MySubClass>).BaseType);
```

[VB]

Using the same definition as in the example above, it is also true that `SomeClass(Of MyBaseClass)` is not the base type of `SomeClass(Of MySubClass)`:

```
Debug.Assert(GetType(SomeClass(Of MyBaseClass)) IsNot GetType(SomeClass(Of
MySubClass)).BaseType)
```

[C++]

Using the same definition as in the example above, it is also true that `MyClass<MyBaseClass>` is not the base type of `MyClass<MySubClass>`:

```
Type ^baseType = typeid<MyClass<MyBaseClass ^> ^>;
Type ^subType  = typeid<MyClass<MySubClass  ^> ^>;
Debug::Assert(baseType != subType);
```

This would not be the case if the generic types were contra-variant.

Because generics are not covariant, when overriding a virtual method that returns a generic type parameter, you cannot provide a subtype of that type parameter as the definition of the overriding method:

For example, the following statement does not compile:

[C#]

```
class MyBaseClass<T>
{
```

```
    public virtual T MyMethod()
    {...}
}
class MySubClass<T,U> : MyBaseClass<T> where T : U
{
    //Invalid definition:
    public override U MyMethod()
    {...}
}
```

[VB]

```
Class MyBaseClass(Of T)
  Public Overridable Function MyMethod() As T
  …
  End Function
End Class

Class MySubClass(Of T As U, U)
     Inherits MyBaseClass(Of T)
End Class
```

[C++]

C++ doesn't allow a generic type to be used as a constraint.

That said, constraints are covariant. For example, you can satisfy a constraint using a sub type of the constraint's type:

[C#]

```
class MyBaseClass
{}
class MySubClass : MyBaseClass
{}
class MyClass<T> where T : MyBaseClass
{}

MyClass<MySubClass> obj = new MyClass<MySubClass>();
```

[VB]

```
Class MyBaseClass
    ...
End Class
Class MySubClass
    Inherits MyBaseClass
    ...
End Class
Class SomeClass(Of T As MyBaseClass)
    ...
End Class
Dim obj As New SomeClass(Of MySubClass)()
```

[C++]

```
ref class MyBaseClass
{};
```

```
ref class MySubClass : MyBaseClass
{};
generic <typename T> where T : MyBaseClass
ref class MyClass
{};

MyClass<MySubClass ^> ^obj = gcnew MyClass<MySubClass ^>;
```

You can even further restrict constraints this way:

[C#]

```
class BaseClass<T>  where T : IMyInterface
{}
interface IMyOtherInterface : IMyInterface
{}

class SubClass<T> : BaseClass<T> where T : IMyOtherInterface
{}
```

[VB]

```
Class BaseClass(Of T As IMyInterface)
    ...
End Class
Interface IMyOtherInterface
    Inherits IMyInterface
    ...
End Interface
Class SubClass(Of T As IMyOtherInterface)
    Inherits BaseClass(Of T)
    ...
End Class
```

[C++]

```
interface class IMyInterface
{};
generic <typename T> where T : IMyInterface
ref class BaseClass
{};
interface class IMyOtherInterface : IMyInterface
{};
generic <typename T> where T : IMyOtherInterface
ref class SubClass : BaseClass<T>
{};
```

Finally, generics are invariant, because there is no relationship between two generic types with different type arguments, even if those type arguments do have an is-as relationship, for example, `List<int>` has nothing to do with `List<object>`, even though an `int` is an `object`.

What can define generic type parameters? What types can be generic?

Classes, interfaces, structures and delegates, can all be generic types. Here are a few examples from the .NET Framework:

[C#]

```
public interface IEnumerator<T> : IEnumerator,IDisposable
```

```
{
   T Current{get;}
}

public class List<T> : IList<T> //More interfaces
{
   public void Add(T item);
   public bool Remove(T item);
   public T this[int index]{get;set;}
   //More members
}

public struct KeyValuePair<K,V>
{
   public KeyValuePair(K key,V value);
   public K Key;
   public V Value;
}

public delegate void EventHandler<E>(object sender,E e) where E : EventArgs;
```

[VB]

```
Public Interface IEnumerator(Of T)
    Inherits IDisposable , IEnumerator
    ReadOnly Property current As T
End Interface

Public Class list(Of T)
    Inherits IList(Of T)'More interfaces

    Public Sub Add(ByVal item As T)
    Public Function Remove(ByVal item As T) As Boolean
    ' More members
End Class

Public Struct KeyValuePair(Of K, V)
    Public Sub New(key As K, value As V)
    Public Key As K
    Public Value As V
End Structure
Public Delegate Sub EventHandler(Of E As EventArgs)(ByVal sender As Object, ByVal e
As E)
```

[C++]

```
generic <typename T>
public interface class IEnumerator : IEnumerator,IDisposable
{
   property T Current { T get(); }
};

generic <typename T>
public ref class List : IList<T> //More interfaces
{
public:
   void Add(T item);
   bool Remove(T item);
   property T default[] { T get(int index); void set(int index, T value); }
```

```
   //More memebers
};
generic <typename K, typename V>
public ref struct KeyValuePair
{
public:
   KeyValuePair(K key, V value);
   K Key;
   V Value;
};

generic <typename T> where T: EventArgs
public delegate void EventHandler(Object ^sender, T e);
```

In addition, both static and instance methods can rely on generic type parameters, independent of the types that contain them:

[C#]

```
public sealed class Activator : _Activator
{
   public static T CreateInstance<T>();
   //Additional memebrs
}
```

[VB]

```
Public NotInheritable Class Activator
      Implements _Activator

    Public Shared Function CreateInstance(Of T)() As T
    ' Additional members.
End Class
```

[C++]

```
public ref class Activator sealed : _Activator
{
   public: generic <typename T>
           static T CreateInstance();
   //Additional members
};
```

Enumerations on the other hand cannot define type parameters, and the same goes for attributes.

Can methods define generic type parameters? How do I call such methods?

Yes. Both instance and static methods can define generic type parameters, and do so independently of their containing class. For example:

[C#]

```
public class MyClass
{
   public void MyInstanceMethod<T>(T t)
   {...}
```

```
    public static void MyStaticMethod<T>(T t)
    {...}
}
```

[VB]

```
Public Class SomeClass
    Public Sub MyInstanceMethod(Of T)(ByVal value As T)
        ...
    End Sub
    Public Shared Sub MySharedMethod(Of T)(ByVal value As T)
        ...
    End Sub
End Class
```

[C++]

```
public ref class MyClass
{
public:
   generic <typename T>
   void MyInstanceMethod (T t)
   {...}
   generic <typename T>
   static void MyStaticMethod (T t)
   {...}
};
```

The benefit of a method that defines generic type parameters is that you can call the method passing each time different parameter types, without ever overloading the method. When you call a method that defines generic type parameters, you need to provide the type arguments at the call site:

[C#]

```
MyClass obj = new MyClass();
obj.MyInstanceMethod<int>(3);
obj.MyInstanceMethod<string>("Hello");

MyClass.MyStaticMethod<int>(3);
MyClass.MyStaticMethod<string>("Hello");
```

[VB]

```
Dim obj As New SomeClass()
obj.MyInstanceMethod(Of Integer)(3)
obj.MyInstanceMethod(Of String)("Hello")
SomeClass.MySharedMethod(Of Integer)(3)
SomeClass.MySharedMethod(Of String)("Hello")
```

[C++]

```
MyClass ^obj = gcnew MyClass;
obj->MyInstanceMethod<int>(3);
obj->MyInstanceMethod<String ^>("Hello");

MyClass::MyStaticMethod<int>(3);
MyClass::MyStaticMethod<String ^>("Hello");
```

If type-inference is available, you can omit specifying the type arguments at the call site:

[C#]

```
MyClass obj = new MyClass();
obj.MyInstanceMethod(3);
obj.MyInstanceMethod("Hello");

MyClass.MyStaticMethod(3);
MyClass.MyStaticMethod("Hello");
```

[VB]

```
Dim obj As New SomeClass()
obj.MyInstanceMethod(3)
obj.MyInstanceMethod("Hello")
SomeClass.MySharedMethod(3)
SomeClass.MySharedMethod("Hello")
```

[C++]

```
MyClass ^obj = gcnew MyClass;
obj->MyInstanceMethod(3);
obj->MyInstanceMethod(gcnew String("Hello"));

MyClass::MyStaticMethod(3);
MyClass::MyStaticMethod(gcnew String("Hello"));
```

Can I derive from a generic type parameter?

You cannot define a class that derives from its own generic type parameter:

[C#]

```
public class MyClass<T> : T //Does not compile
{...}
```

[VB]

```
Public Class SomeClass(Of T)
    Inherits T ' Does not compile
    ...
End Class
```

[C++]

```
generic <typename T>
public ref class MyClass : T //Does not compile
{...};
```

What is a generic type inference?

Generic type inference is the compiler's ability to infer which type arguments to use with a generic method, without the developer having to specify it explicitly. For example, consider the following definition of generic methods:

[C#]

```
public class MyClass
{
   public void MyInstanceMethod<T>(T t)
   {...}
   public static void MyStaticMethod<T>(T t)
   {...}
```

```
}
```

[VB]

```
Public Class SomeClass
    Public Sub MyInstanceMethod(Of T)(ByVal value As T)
        ...
    End Sub
    Public Shared Sub MySharedMethod(Of T)(ByVal value As T)
        ...
    End Sub
End Class
```

[C++]

```
public class MyClass
{
public:
   generic <typename T>
   void MyInstanceMethod(T t) {...}
   generic <typename T>
   static void MyStaticMethod (T t) {...}
};
```

When invoking these methods, you can omit specifying the type arguments for both the instance and the static methods:

[C#]

```
MyClass obj = new MyClass();
obj.MyInstanceMethod(3); //Compiler infers T as int
obj.MyInstanceMethod("Hello");//Compiler infers T as string

MyClass.MyStaticMethod(3); //Compiler infers T as int
MyClass.MyStaticMethod("Hello");//Compiler infers T as string
```

[VB]

```
Dim obj As New SomeClass()
obj.MyInstanceMethod(3) ' Compiler infers T as int
obj.MyInstanceMethod("Hello") ' Compiler infers T as String
SomeClass.MySharedMethod(3) ' Compiler infers T as Integer
SomeClass.MySharedMethod("Hello") ' Compiler infers T as string
```

[C++]

```
MyClass ^obj = gcnew MyClass;
obj->MyInstanceMethod(3); //Compiler infers T as int
MyClass::MyStaticMethod(3); //Compiler infers T as int
```

Note that type inferring is possible only when the method takes an argument of the inferred type arguments. For example, in the `CreateInstance<T>()` method of the `Activator` class, defined as:

[C#]

```
public sealed class Activator : _Activator
{
   public static T CreateInstance<T>();
   //Additional memebrs
}
```

[VB]

```
Public NotInheritable Class Activator
      Implements _Activator

    Public Shared Function CreateInstance(Of T)() As T
    ' Additional members.
End Class
```

[C++]

```
public ref class Activator sealed : _Activator
{
public:
   generic <typename T>
   static T CreateInstance ();
   //Additional members
};
```

type inference is not possible, and you need to specify the type arguments at the call site:

[C#]

```
class MyClass
{...}
MyClass obj = Activator.CreateInstance<MyClass>();
```

[VB]

```
Public Class SomeClass
    ...
End Class
Dim obj As SomeClass = activator.createInstance(Of SomeClass)()
```

[C++]

```
ref class MyClass
{...};
MyClass ^obj = Activator::CreateInstance<MyClass ^>();
```

Note also that you cannot rely on type inference at the type level, only at the method level. In the following example, you must still provide the type argument `T` even though the method takes a `T` parameter:

[C#]

```
public class MyClass<T>
{
   public static void MyStaticMethod<U>(T t,U u)
   {...}
}
MyClass<int>.MyStaticMethod(3,"Hello");//No type inference for the integer
```

[VB]

```
Public Class SomeClass(Of T)
    Public Shared sub MySharedMethod(Of U)(ByVal item As T, ByVal uu As U)
        ...
    End Sub
End Class
SomeClass(Of Integer).MySharedMethod(3, "Hello")'No type inference for the integer
```

[C++]

```
generic <typename T>
public ref class MyClass
{
   public: generic <typename U> static void MyStaticMethod (T t,U u)
   {...}
};
MyClass<int>::MyStaticMethod(3, 22.7);//No type inference for the integer
```

What are constraints?

Constraints allow additional contextual information to be added to the type parameters of generic types. The constraints limit the range of types that are allowed to be used as type arguments, but at the same time, they add information about those type parameters. Constraints ensure that the type arguments specified by the client code are compatible with the generic type parameters the generic type itself uses. Meaning, constraints prevent the client from specifying types as type arguments that do not offer the methods, properties, or members of the generic type parameters that the generic type relies upon.

After applying a constraint you get IntelliSense reflecting the constraints when using the generic type parameter, such as suggesting methods or members from the base type.

There are three types of constraints:

Derivation constraint indicates to the compiler that the generic type parameter derives from a base type such an interface or a particular base class. For example, in the following example, the linked list applies a constraint of deriving from `IComparable<T>` on its generic type parameter. This is required so that you could implement a search. sorting or indexing functionality on the list:

[C#]

```
class Node<K,T>
{
   public K Key;
   public T Item;
   public Node<K,T> NextNode;
}

public class LinkedList<K,T> where K : IComparable<K>
{
   Node<K,T> m_Head;

   public T this[K key]
   {
      get
      {
         Node<K,T> current = m_Head;
         while(current.NextNode != null)
         {
            if(current.Key.CompareTo(key) == 0)
               break;
            else
               current = current.NextNode;
         }
         return current.Item;
```

```
      }
   }
   //Rest of the implementation
}
```

[VB]

```
Class Node(Of K, T)
    Public Key As K
    Public Item As T
    Public NextNode As Node(Of K, T)
End Class

Public Class LinkedList(Of K As IComparable(Of K), T)
    Dim m_Head As Node(Of K, T)
    Public ReadOnly Property Item(ByVal key As K) As T
        Get
            Dim current As Node(Of K, T) = m_Head
            While current.NextNode IsNot Nothing
                If (current.Key.CompareTo(key) = 0) Then
                    Exit While
                Else
                    current = current.NextNode
                End If
            End While
            Return current.item
        End Get
    End Property
    ' Rest of the implementation
End Class
```

[C++]

```
generic <typename K, typename T>
ref class Node
{
public: K Key;
        T Item;
        Node<K,T> ^NextNode;
};
generic <typename K, typename T> where K : IComparable<K>
public ref class LinkedList
{
public:
   Node<K,T> ^m_Head;
public:
   property T default[]
   {
      T get(K key)
      {
         Node<K,T> ^current = m_Head;
         while(current->NextNode)
         {
            if(current->Key->CompareTo(key)==0)
               break;
            else
               current = current->NextNode;
         }
         return current->Item;
```

```
        }
    }
    //Rest of the implementation
};
```

You can provide constraints for every generic type parameter that your class declares, for example:

[C#]

```
public class LinkedList<K,T> where K : IComparable<K>
                             where T : ICloneable
```

[VB]

```
Public Class LinkedList(Of K As IComparable(Of K), T As ICloneable)
    ...
End Class
```

[C++]

```
generic <typename K, typename T> where K : IComparable<K>
                                 where T : ICloneable
public ref class LinkedList
{ ... };
```

You can have a base class constraint, meaning, stipulating that the generic type parameter derives from a particular base class:

[C#]

```
public class MyBaseClass
{...}
public class MyClass<T> where T : MyBaseClass
{...}
```

[VB]

```
Public Class MyBaseClass
    ...
End Class
Public Class SomeClass(Of T As MyBaseClass)
    ...
End Class
```

[C++]

```
public ref class MyBaseClass
{...};
generic <typename T> where T : MyBaseClass
public ref class MyClass
{...};
```

However, you can only use one base class at most in a constraint because neither C#, VB or managed C++ support multiple inheritance of implementation. Obviously, the base class you constrain to cannot be a sealed class, and the compiler enforces that. In addition, you cannot constrain `System.Delegate` or `System.Array` as a base class.

You can constrain both a base class and one or more interfaces, but the base class must appear first in the derivation constraint list:

[C#]

```
public class LinkedList<K,T> where K : MyBaseKey,IComparable<K>
{...}
```

[VB]

```
Public Class LinkedList(Of K As {MyBaseKey, IComparable(Of K)}, T)
    ...
End Class
```

[C++]

```
generic <typename K, typename T> where K : MyBaseKey, IComparable<K>
public class LinkedList
{...};
```

The *constructor constraint* indicates to the compiler that the generic type parameter exposes a default public constructor (a public constructor with no parameters). For example:

[C#]

```
class Node<K,T> where K : new()
                where T : new()
{
   public K Key;
   public T Item;
   public Node<K,T> NextNode;

   public Node()
   {
      Key      = new K(); //Compiles because of the constraint
      Item     = new T(); //Compiles because of the constraint
      NextNode = null;
   }
   //Rest of the implementation
}
```

[VB]

```
Class Node(Of K As New, T As New)
    Public Key  As K
    Public Item As T
    Public NextNode As Node(Of K, T)
    Public Sub New()
        Key  = New K()' Compiles because of the constraint
        Item = New T()' Compiles because of the constraint
        NextNode = Nothing
    End Sub
    ' Rest of the implementation.
End Class
```

[C++]

[Add C++ sample. The C++ team claims support for where T = gcnew() is forthcoming. JL]

You can combine the default constructor constraint with derivation constraints, provided the default constructor constraint appears last in the constraint list:

[C#]

```
public class LinkedList<K,T> where K : IComparable<K>,new()
                                  where T : new()
{...}
```

[VB]

```
Public Class LinkedList(Of K As {IComparable(Of K), New}, T As New)
    ...
End Class
```

The *reference* and *value type constraint* is used to constrain the generic type parameter to be a value or a reference type. For example, you can constrain a generic type parameter to be a value type (such as an `int`, a `bool`, and `enum`, or any structure):

[C#]

```
public class MyClass<T> where T : struct
{...}
```

[VB]

```
Public Class SomeClass(Of T As Structure)
    ...
End Class
```

[C++]

[Add C++ sample. The C++ team claims support for this is forthcoming. JL]

Similarly, you can constrain a generic type parameter to be a reference type (a class):

[C#]

```
public class MyClass<T> where T : class
{...}
```

[VB]

```
Public Class SomeClass(Of T As Class)
    ...
End Class
```

[C++]

[Add C++ sample. The C++ team claims support for this is forthcoming. JL]

The reference and value type constraint cannot be used with a base class constraint, but it can be combined with any other constraint. When used, the value/reference type constraint must appear first in the constraint list.

It is important to note that although constraints are optional, they are often essential when developing a generic type. Without constraints, the compiler follows the more conservative, type-safe approach and only allows access to `object`-level functionality in your generic type parameters. Constraints are part of the generic type metadata so that the client-side compiler can take advantage of them as well. The client-side compiler only

allows the client developer to use types that comply with the constraints, thus enforcing type safety.

What can I not use constraints with?

You can only place a derivation constraint on a type parameter (be it an interface derivation or a single base class derivation). In C# and VB, you can also use a default constructor constraint and a value or reference type constraint. While everything else is implicitly not allowed, it is worth mentioning the specific cases that are not possible:

- You cannot constrain a generic type to have any specific parameterized construct.
- You cannot constrain a generic type to derive from a sealed class.
- You cannot constrain a generic type to derive from a static class.
- You cannot constrain a public generic type to derive from another internal type.
- You cannot constrain a generic type to have a specific method, be it a static or an instance method.
- You cannot constrain a generic type to have a specific public event.
- You cannot constrain a generic type parameter to derive from `System.Delegate` or `System.Array`.
- You cannot constrain a generic type parameter to be serializable.
- You cannot constrain a generic type parameter to be COM-visible.
- You cannot constrain a generic type parameter to have any particular attribute.
- You cannot constrain a generic type parameter to support any specific operator. There is therefore no way to compile the following code:

[C#]

```
public class Calculator<T>
{
   public T Add(T argument1,T argument2)
   {
      return argument1 + argument2; //Does not compile
   }
   //Rest of the methods
}
```

[VB]

```
Public Class calculator(Of T)
    Public Function add(ByVal argument1 As T, ByVal argument2 As T) As T
        Return argument1 + argument2
        ' The preceding statement does not compile.
    End Function
    ' Rest of the methods.
End Class
```

[C++]

```
generic <typename T>
public ref class Calculator
{
public: T Add(T argument1,T argument2)
   {
```

```
        return argument1 + argument2; //Does not compile
    }
    //Rest of the methods
};
```

Why cannot I use enums, structs, or sealed classes as generic constraints

You cannot constraint a generic type parameter to derive from a non-derivable type. For example, the following does not compile:

[C#]

```
public sealed class MySealedClass
{...}
public class MyClass<T> where T : MySealedClass //Does not compile
{...}
```

[VB]

```
Public NotInheritable Class MySealedClass
    ...
End Class
Public Class SomeClass(Of T As MySealedClass
' The preceding statement does not compile.
    ...
End Class
```

[C++]

```
public ref class MySealedClass sealed
{...};
generic <typename T> where T : MySealedClass
public ref class MyClass //Does not compile
{...};
```

The reason is simple: The only type arguments that could possibly satisfy the above constraint is the type `MySealedClass` itself, making the use of generics redundant. For this very reason, all other non-derivable types such as structures and enums are not allowed in constraints.

Is code that uses generics faster than code that does not?

The answer depends on the way the non-generic code is written. If the code is using objects as the amorphous containers to store items, then various benchmarks have shown that in intense calling patterns, generics yield on average 100% performance improvement (that is, three times as fast) when using value types, and some 50% performance improvement when using reference types.

If the non-generic code is using type-specific data structures, then there is no performance benefit to generics. However, such code is inherently very fragile. Writing a type-specific data structure is a tedious, repetitive, and error-prone task. When you fix a defect in the data structure, you have to fix it not just in one place, but in as many places as there are type-specific duplicates of what essentially is the same data structure.

Is an application that uses generics faster than an application that does not?

Depending on the application of course, but generally speaking, in most real-life applications, bottle necks such as I/O will mask out any performance benefit from

generics. The real benefit of generics is not performance but rather type safety and productivity.

How are generics similar to classic Visual C++ templates?

Generics are similar in concept to classic C++ templates: both allow data structures or utility classes to defer to the client the actual types to use, and both offer productivity and type-safety benefits.

How are generics different from classic Visual C++ templates?

There are two main differences: in the programming model and in the underlying implementation. In the programming model, .NET generics can provide enhanced safety compared to classic Visual C++ templates. .NET generics have the notion of constraints, which gives you added type safety. On the other hand, .NET generics offer a more restrictive programming model – there are quite a few things that generics cannot do, such as using operators, because there is no way to constraint a type parameter to support an operator. This is not the case in classic Visual C++ templates where you can apply any operator you like on the type parameters. At compile time, the classic Visual C++ compiler will replace all the type parameters in the template with your specified type, and any incompatibility is usually discovered then.

Both templates and generics can incur some code bloat, and both have mechanisms to limit that bloat. Instantiating a template with a specific set of types instantiates only the methods actually used; and then all methods that result in identical code are automatically merged by the compiler which prevents needless duplication. Instantiating a generic with a specific set of types instantiates all of its methods, but only once for *all* reference type arguments; bloat comes only from value types, because the CLR instantiates a generic separately once for *each* value type argument. Finally, .NET generics allow you to ship binaries, while C++ templates require you to share some code with the client.

What is the difference between using generics and using interfaces (or abstract classes)?

Interfaces and generics serve different purposes. Interfaces are about defining a contract between a service consumer and a service provider. As long as the consumer programs strictly against the interface (and not a particular implementation of it), it can use any other service provider that supports the same interface. This allows switching service providers without affecting (or with minimum effect on) the client's code. The interface also allows the same service provider to provide services to different clients. Interfaces are the cornerstone of modern software engineering, and are used extensively in past and future technologies, from COM to .NET to Indigo and SOA.

Generics are about defining and implementing a service without committing to the actual types used. As such, interfaces and generics are not mutually exclusive. Far from it, they compliment each other. You can and you should combine interfaces and generics.

For example, the interface `ILinkedList<T>` defined as:

[C#]

```
public interface ILinkedList<T>
{
   void AddHead(T item);
```

```
    void RemoveHead(T item);
    void RemoveAll();
}
```

[VB]

```
Public Interface ILinkedList(Of T)
    Sub AddHead(ByVal item As T)
    Sub RemoveHead(ByVal item As T)
    Sub RemoveAll()
End Interface
```

[C++]

```
generic <typename T>
public interface class ILinkedList
{
    void AddHead(T item);
    void RemoveHead(T item);
    void RemoveAll();
};
```

Can be implemented by any linked list:

[C#]

```
public class LinkedList<T> : ILinkedList<T>
{...}

public class MyOtherLinkedList<T> : ILinkedList<T>
{...}
```

[VB]

```
Public Class LinkedList(Of T)
    Implements ILinkedList(Of T)
    ...
End Class
Public Class MyOtherinkedList(Of T)
    Implements ILinkedList(Of T)
    ...
End Class
```

[C++]

```
generic <typename T>
public ref class LinkedList : ILinkedList<T>
{...};

generic <typename T>
public ref class MyOtherLinkedList : ILinkedList<T>
{...};
```

You can now program against `ILinkedList<T>`, using both different implementations and different type arguments:

[C#]

```
ILinkedList<int> numbers  = new LinkedList<int>();
ILinkedList<string> names = new LinkedList<string>();

ILinkedList<int> moreNumbers = new MyOtherLinkedList<int>();
```

[VB]

```
Dim numbers As ILinkedList(Of Integer) = New LinkedList(Of Integer)()
Dim names As ILinkedList(Of String) = New LinkedList(Of String)()
Dim moreNumbers As ILinkedList(Of Integer) = New MyOtherLinkedList(Of Integer)()
```

[C++]

```
ILinkedList<int> ^numbers    = gcnew LinkedList<int>;
ILinkedList<String ^> ^names = gcnew LinkedList<String ^>;

ILinkedList<int> ^moreNumbers = gcnew MyOtherLinkedList<int>();
```

How are generics implemented?

Generics have native support in IL and the CLR itself. When you compile generic server-side code, the compiler compiles it into IL, just like any other type. However, the IL only contains parameters or place holders for the actual specific types. In addition, the metadata of the generic server contains generic information such as constraints.

The client-side compiler uses that generic metadata to support type safety. When the client provides a type arguments, the client's compiler substitutes the generic type parameter in the server metadata with the specified type. This provides the client's compiler with type-specific definition of the server, as if generics were never involved. At run time, the actual machine code produced depends on whether the specified types are value or reference type. If the client specifies a value type, the JIT compiler replaces the generic type parameters in the IL with the specific value type, and compiles it to native code. However, the JIT compiler keeps track of type-specific server code it already generated. If the JIT compiler is asked to compile the generic server with a value type it has already compiled to machine code, it simply returns a reference to that server code. Because the JIT compiler uses the same value-type-specific server code in all further encounters, there is no code bloating.

If the client specifies a reference type, then the JIT compiler replaces the generic parameters in the server IL with `object`, and compiles it into native code. That code will be used in any further requests for a reference type instead of a generic type parameter. Note that this way the JIT compiler only reuses actual code. Instances are still allocated according to their size off the managed heap, and there is no casting.

Why can't I use operators on naked generic type parameters?

The reason is simple – Since there is no way to constrain a generic type parameter to support an operator, there is no way the compiler can tell whether the type specified by the client of the generic type will support the operator.

Consider for example the following code:

[C#]

```
class Node<K,T>
{
   public K Key;
   public T Item;
   public Node<K,T> NextNode;
}

public class LinkedList<K,T>
```

```
{
   Node<K,T> m_Head;

   public T this[K key]
   {
      get
      {
         Node<K,T> current = m_Head;
         while(current.NextNode != null)
         {
            if(current.Key == key)) //Does not compile
               break;
            else
               current = current.NextNode;
         }
         return current.Item;
      }
   }
   //Rest of the implementation
}
```

[VB]

```
Class Node(Of K, T)
    Public Key As K
    Public Item As T
    Public NextNode As Node(Of K, T)
End Class
```

```
<DefaultMember("Item")> _
Public Class LinkedList(Of K, T)

    Dim m_Head As Node(Of K, T)

    Public ReadOnly Property Item(ByVal key As K) As T
        Get
            Dim current As Node(Of K, T) = m_Head
            While current.NextNode IsNot Nothing
                If current.key = key Then
            ' The preceding statement does not compile.
                    Exit While
                Else
                    current = current.NextNode
                End If
            End While
            Return current.item
        End Get
    End Property
    ' Rest of the implementation
End Class
```

[C++]

```
generic <typename K, typename T>
ref class Node
{
public: K Key;
        T Item;
        Node<K,T> ^NextNode;
```

```
};

generic <typename K, typename T>
public ref class LinkedList
{
public:
   Node<K,T> ^m_Head;
public:
   property T default[]
   {
      T get(K key)
      {
         Node<K,T> ^current = m_Head;
         while(current->NextNode)
         {
            if(current->Key == key)) //Does not compile
               break;
            else
               current = current->NextNode;
         }
         return current->Item;
      }
   }
   //Rest of the implementation
};
```

The compiler will refuse to compile this line:

[C#]

```
if(current.Key == key))
```

[VB]

```
If current.key = key Then
```

[C++]

```
if(current->Key == key))
```

Because it has no way of knowing whether the type the consumer will specify will support the == operator.

When can I use operators on generic type parameters?

You can use an operator (or for that matter, any type-specific method) on generic type parameters if the generic type parameter is constrained to be a type that supports that operator. For example:

[C#]

```
class MyOtherClass
{
   public static MyOtherClass operator+(MyOtherClass lhs,MyOtherClass rhs)
   {
      MyOtherClass product = new MyOtherClass();
      product.m_Number = lhs.m_Number + rhs.m_Number;
      return product;
   }
   int m_Number;
```

```
   //Rest of the class
}

class MyClass<T> where T : MyOtherClass
{
   MyOtherClass Sum(T t1,T t2)
   {
      return t1 + t2;
   }
}
```

[VB]

```
Class MyOtherClass
   Public Shared Function op_Addition(ByVal lhs As MyOtherClass,
                                      ByVal rhs As MyOtherClass) As MyOtherClass
      Dim product As New MyOtherClass
      product.m_Number =  lhs.m_Number + rhs.m_Number
      return product
   End Function
   Private m_Number As Integer
End Class

Class SomeClass(Of T As MyOtherClass)
   Private Function Sum(ByVal t1 As T, ByVal t2 As T) As MyOtherClass
      Return (t1 + t2)
   End Function
End Class
```

[C++]

[Add C++ sample JL]

Can I use generic attributes?

You cannot define generic attributes:

[C#]

```
//This is not possible:
class MyAttribute<T>: Attribute
{...}
```

[VB]

```
' The following declaration is not possible.
Public Class MyAttribute(Of T)
    Inherits Attribute
    ...
End Class
```

[C++]

```
//This is not possible:
generic <typename T>
ref class MyAttribute: Attribute
{...};
```

[The C++ team claims this is possible, although I could not compile it, please verify JL]

However, nothing prevents you from using generics internally, inside the attribute's implementation.

Are generics CLS Compliant?

Yes. With the release of .NET 2.0, generics will become part of the CLS.

.NET Framework

Which versions of the .NET Framework support generics

Generics are only supported on version 2.0 and above of the Microsoft .NET framework, as well as version 2.0 of the compact framework.

Can I use generics in Web services?

Unfortunately, no. Web services have to expose a WSDL-based contract. Such contracts are always limited by the expressiveness of the message format being used. For example, HTTP-GET based web services only support primitive types such as `int` or `string`, but not complex types like a `DataSet`. SOAP-based web services are more capable, but SOAP has no ability to represent generic type parameters. As a result, at present, you cannot define web services that rely on generic types. That said, you can define .NET web services that rely on closed constructed generic types, for example:

[C#]

```
public class MyWebService
{
   [WebMethod]
   public List<string> GetCities()
   {
      List<string> cities = new List<string>();
      cities.Add("New York");
      cities.Add("San Francisco");
      cities.Add("London");
      return cities;
   }
}
```

[VB]

```
Public Class MyWebService
    <WebMethod>
    Public Function GetCities() As List(Of String)
        Dim cities As New List(Of String)()
        cities.add("New York")
        cities.add("San Francisco")
        cities.add("London")
        Return cities
    End Function
End Class
```

[C++]

```
public ref class MyWebService
{
public:
   [WebMethod]
```

```
   List<String ^> ^ GetCities()
   {
      List<String ^> ^cities = gcnew List<String ^>;
      cities->Add("New York");
      cities->Add("San Francisco");
      cities->Add("London");
      return cities;
   }
}
```

In the above example, `List<string>` will be marshaled as an array of strings.

Can I use generics in Enterprise Services?

Unfortunately, no. All methods and interfaces on a `ServicedComponent`-derived class must be COM-visible. The COM type system is IDL, and IDL does not support type parameters.

Can I use generics in Indigo?

Unfortunately, no. SOAP has no ability to represent generic type parameters, and so all methods and interfaces on an indigo service contract or service class can only use primitive types such as integers or strings, or specific known types that provide a data contract. As a result, at present, you cannot define Indigo services that rely on generic types, that is, services that leave it up to the service consumer to specify the types to use when invoking the service.

Can I use generics in .NET Remoting?

Yes. You can expose generic types as remote objects, for example:

[C#]

```
public class MyRemoteClass<T> : MarshalByRefObject
{...}
Type serverType = typeof(MyRemoteClass<int>);

RemotingConfiguration.RegisterWellKnownServiceType(serverType,
                                                   "Some URI",
                                                  WellKnownObjectMode.SingleCall);
```

[VB]

```
Public Class MyRemoteClass(Of T)
    Inherits MarshalByRefObject
    ...
End Class

Dim serverType As Type = GetType(MyRemoteClass(Of Integer))
RemotingConfiguration.RegisterWellKnownServiceType(serverType, _
                                                   "Some URI", _
                                                   WellKnownObjectMode.SingleCall)
```

[C++]

```
generic <typename T>
public ref class MyRemoteClass : MarshalByRefObject
{...};
Type ^serverType = typeid<MyRemoteClass<int> ^>;
```

```
RemotingConfiguration::RegisterWellKnownServiceType(serverType,
                                                    "Some URI",
                                                   WellKnownObjectMode::SingleCall);
```

Note that the specific type arguments used must be a marshalable type, that is, either serializable or derived from `MarshalByRefObject`. Consequently, a generic remote type will typically place a derivation constraint from `MarshalByRefObject` on its generic type parameters when expecting reference type parameters:

[C#]

```
public class MyRemoteClass<T> : MarshalByRefObject where T : MarshalByRefObject
{...}
```

[VB]

```
Public Class MyRemoteClass(Of T As MarshalByRefObject)
    Inherits MarshalByRefObject
    ...
End Class
```

[C++]

```
generic <typename T> where T : MarshalByRefObject
public ref class MyRemoteClass: MarshalByRefObject
{...};
```

To administratively register a generic type, provide the type arguments in double square brackets.

For example, to register the class MyRemoteClass<T> with an integer, you should write:

```
<service>
   <wellknown type="MyRemoteClass[[System.Int32]],ServerAssembly"
              mode="SingleCall" objectUri="Some URI"/>
</service>
```

The double square brackets is required in case you need to specify multiple type arguments, in which case, each type arguments would be encased in a separate pair of brackets, separated by a comma. For example, to register the class MyRemoteClass<T,U> with an integer and a string, you would write:

```
<service>
   <wellknown type="MyRemoteClass[[System.Int32],[System.String]],
                    ServerAssembly" mode="SingleCall" objectUri="Some URI"/>
</service>
```

Creating a new instance of generic remote objects is done just as with non-generic remote objects.

Can I use Visual Studio 2003 or the .NET Framework 1.1 to create generics?

Unfortunately, no. Generics are only supported on version 2.0 and above of the Microsoft .NET framework. Code that relies on generics must run on version 2.0 of the CLR. Because of the way the CLR version unification works, a run-time process can only load a single version of the CLR. Consequently, a process that loaded version 1.1 of the CLR cannot use generic types. If you must use generic types from .NET 1.1, you can use the following work-around: First, wrap the generic types with object-based types (at the

expense of course of the benefits of using generics). Next, load the wrapper classes in a separate process which loads version 2.0 of the CLR, and provide remote access to the wrapper classes to legacy clients in process that use version 1.1 of the CLR. For remote communication you can use any number of cross-process communication mechanisms, such as Remoting, Enterprise Services, sockets, etc.

What environment do I need to use generics?

To deploy and run code that uses generics you need version 2.0 or higher of the .NET runtime.

Can I use generics on the Compact Framework?

Yes. The .NET Compact Framework version 2.0 supports generics. Like most other things with the .NET Compact Framework, the generics support is very close but not exactly the same as the normal .NET Framework, due to performance and schedule constrains. You can use generics with both C# and Visual Basic for the compact framework. The compact framework does apply certain limitations on generics, the notable ones are:

The compact framework does not verify constraints are runtime, only at compile time.

You can only have up to 8 generic type parameters per generic type.

You cannot use reflection on unbounded generic types.

Which .NET languages support generics and how?

Both C# 2.0 and Visual Basic 2005 support defining and consuming generics. Visual C++ 2005 also supports generics in addition to classic C++ templates. Visual J# 2005 supports consuming generic types but not defining them. At present, it is not known of other vendors besides Microsoft that added generics support for their languages.

Where does the .NET Framework itself use generics?

Version 2.0 of the .NET Framework makes use of generics in three main areas: The `System` namespace added a large set of static generic methods to the `Array` type. These methods automate and streamline common manipulations of and interactions with arrays. The `System` namespace also defined a number of generic utility delegates, which are used by the `Array` type and the `List<T>` class, but can be used freely in other contexts as well. In addition, `System` provides support for nullable types. The `System` namespace defines the `IComparable<T>` interface and the `EventHandler<E>` delegate, both generic reincarnations of their non-generic predecessors. The `System` namespace also defines the `IEquatable<T>` interface, used to check for equality of two values. The `System` namespace defines the `ArraySegment<T>` used to allocate a strongly typed portion of an array.

The `System.Collections.Generic` namespace defines generic collection interfaces, collections and iterator classes, similar to the old, non generic ones available in the `System.Collections` namespace. The `System.Collections.Generic` namespace also defines a few generic helper classes and structures.

The System.ComponentModel namespace defines the class BindingList<T>. A binding list is used very similar to a mere generic list, except it can fire events notifying interested parties about changes to its state.

The System.Collections.ObjectModel namespace defines a few types such as Collection<T> that can be used as base types for custom collections.

Finally, all the types that supported IComparable in .NET 1.1 support IComparable<T> and IEquatable<T> in .NET 2.0. This enables you to use common types for keys, such as int, string, Version, Guid, DateTime, and so on.

What are the generic collection classes?

The System.Collections.Generic namespace contains the majority of the new generic collections. These collections are by and large the generic reincarnation of the collections available in the System.Collections namespace. For example, there is a generic Stack<T> and a generic Queue<T> classes. The collections in System.Collections.Generic are used in much the same way as their predecessors. In addition, some of the collections where renamed in the process. The Dictionary<K,T> data structure is equivalent to the non-generic HashTable, and the class List<T> is analogous to the non-generic ArrayList. System.Collections.Generic also defines new types that have no equivalent in System.Collections, such as LinkedList<T> and KeyValuePair<K,T>. In addition, The System.Collections.Generic namespace defines generic interfaces such as ICollection<T> and IList<T>. To support generic-based iterators, System.Collections.Generic defines the IEnumerable<T> and IEnumerator<T> interfaces, and these interfaces are supported by all the generic collections. It is important to note that the generic collections can be used by clients that do not rely on generics, because all the generic collections also support the non-generic collection and iteration interfaces (IList, ICollection, IEnumerable). For example, here is the definition of the List<T> class:

[C#]

```
public class List<T> : IList<T>,IList
{...}
```

[VB]

```
Public Class List(Of T)
    Implements IList(Of T),IList
    ...
End Class
```

[C++]

```
generic <typename T>
public ref class List<T> : IList<T>
{...};
```

The System.ComponentModel namespace defines the type BindingList<T>.

[C#]

```
public class BindingList<T> : Collection<T>,
                              IBindingList,ICancelAddNew,IRaiseItemChangedEvents
{
   public event ListChangedEventHandler ListChanged;
   public event AddingNewEventHandler AddingNew;

   public BindingList();
   public BindingList(List<T> list);
   public T AddNew();
   //More members
}
```

[VB]

```
Public class BindingList(Of T)
    Inherits Collection(Of T)
    Implements IBindingList, ICancelAddNew, IRaiseItemChangedEvents
    Public Event ListChangedEventHandler ListChanged
    Public Event AddingNewEventHandler AddingNew
    Public Sub BindingList()
    Public Sub BindingList(ByVal list As List(Of T))
    Public Function AddNew() As T
    ' More members
End Class
```

[C++]

```
generic <typename T>
public ref class BindingList : Collection<T>,IBindingList,
                               ICancelAddNew,IRaiseItemChangedEvents
{
public:
   event ListChangedEventHandler ^ListChanged;
   event ListChangedEventHandler ^ AddingNew;

   public: BindingList();
           BindingList(List<T> ^list);
           T AddNew();
   //More members
}
```

`BindingList<T>` is used similarly to a generic list, except it can fire events notifying interested parties about changes to its state, so you can bind it to user interface controls such as the `ListBox`. You can use `BindingList<T>` directly or you can wrap it around an existing `List<T>`.

The `System.Collections.ObjectModel` namespace defines the types `Collection<T>`, `KeyedCollection<T>`, `ReadOnlyCollection<T>`, and `ReadOnlyCollection<T>` provided as base types for custom providers. Interestingly enough, none of the .NET-provided generic collections actually use these base collections.

Finally, the `System` namespace defines the `ArraySegment<T>` helper structure, which can be used to obtain a generic-based segment of a provided array.

The following table lists the generic collections and their supporting types, including mapping the generic collections to those of `System.Collections` or other namespaces when applicable.

Type	Namespace	Non-Generic Equivalent	Comment
`ArraySegment<T>`	`System`	-	Used to obtain a generic-based segment of a provided array
`BindingList<T>`	`System.ComponentModel`	-	Linked list that fires state changes events
`Collection<T>`	`System.Collections.ObjectModel`	`Collection`	Non abstract base class for other collections
`Comparer<T>`	`System.Collections.Generic`	`Comparer`	Implements `IComparer<T>` and `IComparer`
`Dictionary<K,T>`	`System.Collections.Generic`	`HashTable`	Implements `IDictionary<K,T>`
`EqualityComparer<T>`	`System.Collections.Generic`	-	Abstract class implementing `IEqualityComparer<T>`
`ICollection<T>`	`System.Collections.Generic`	`ICollection`	Count and synchronization for a collection
`IComparer<T>`	`System.Collections.Generic`	`IComparer`	Compares two specified values
`IDictionary<K,T>`	`System.Collections.Generic`	`IDictionary`	Interface for a collection of key/value pairs
`IEnumerable<T>`	`System.Collections.Generic`	`IEnumerable`	Returns an `IEnumerator<T>` object
`IEnumerator<T>`	`System.Collections.Generic`	`IEnumerator`	Iterating over a collection
`IEqualityComparer<T>`	System.Collections.Generic	`IEqualityComparer` (.NET 2.0 only)	Equates two specified values.
`IList<T>`	`System.Collections.Generic`	`IList`	Implemented by list collections or access by index
`KeyedCollection<K,T>`	`System.Collections.ObjectModel`	-	Base class for keyed collections
`KeyValuePair<K,V>`	`System.Collections.Generic`	-	Container for key/value pair
`LinkedList<T>`	`System.Collections.Generic`	-	A true linked list
`LinkedListNode<T>`	`System.Collections.Generic`	-	Used by `LinkedList<T>`, but can be used by custom lists as well.
`List<T>`	`System.Collections.Generic`	`ArrayList`	Impalements `IList<T>` over array
`Queue<T>`	`System.Collections.Generic`	`Queue`	A queue
`ReadOnlyCollection<T>`	`System.Collections.ObjectModel`	`ReadOnlyCollectionBase`	Base class for read-only collections
`SortedDictionary<K,T>`	`System.Collections.Generic`	`SortedList`	Implements `IDictionary<K,T>` over a sorted collection
`SortedList<T>`	`System.Collections.Generic`	`SortedList`	A sorted linked list over an array and a hash table.
`Stack<T>`	`System.Collections.Generic`	`Stack`	A stack

What are the generic delegates?

The `System` namespace defines five new generic delegates. The first is `EventHandler<E>` defined as:

[C#]

```
public delegate void EventHandler<E>(object sender,E e) where E : EventArgs
```

[VB]

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

[C++]

```
generic <typename T> where E : EventArgs
public delegate void EventHandler (Object ^sender, T e);
```

`EventHandler<E>` can be used wherever an event handling method expects an `object` and an `EventArgs`-derived class as parameters. Obviously, that is the case wherever the non-generic `EventHandler` was used in .NET 1.1:

[C#]

```
public delegate void EventHandler(object sender, EventArgs e)
```

[VB]

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

[C++]

```
public delegate void EventHandler(Object ^sender, EventArgs ^e);
```

But in addition, `EventHandler<E>` can be employed instead of all the other delegates that used `EventArgs`-derive class, such as `MouseEventHandler`:

[C#]

```
public class MouseEventArgs : EventArgs
{...}
public delegate void MouseEventHandler(object sender,MouseEventArgs e);

void OnMyMouseEvent(object sender,MouseEventArgs e)
{...}

//Instead of:
MouseEventHandler handler += OnMyMouseEvent;

//You can write:
EventHandler<MouseEventArgs> handler += OnMyMouseEvent;
```

[VB]

```
Public Class MouseEventArgs
    Inherits EventArgs
    ...
```

```
End Class
Public Delegate Sub MouseEventHandler(ByVal sender As Object, ByVal e As
MouseEventArgs)

' Instead of:
Public Class SomeClass
   Event handler As MouseEventHandler

   Public Sub SomeMethod()
      AddHandler handler, AddressOf OnMyMouseEvent
   End Sub

   Sub OnMyMouseEvent(ByVal sender As Object, ByVal e As MouseEventArgs)
      ...
   End Sub

End Class

' You can write:
Public Class SomeClass
   Event handler As EventHandler(Of MouseEventArgs)

   Public Sub SomeMethod()
      AddHandler handler, AddressOf OnMyMouseEvent
   End Sub

   Sub OnMyMouseEvent(ByVal sender As Object, ByVal e As MouseEventArgs)
      ...
   End Sub

End Class
```

[C++]

```
public ref class MouseEventArgs : EventArgs
{...};
public delegate void MouseEventHandler(Object ^sender, MouseEventArgs ^e);

void OnMyMouseEvent(Object ^sender,MouseEventArgs ^e)
{...}

//Instead of:
MouseEventHandler ^handler += gcnew MouseEventHandler(this,
&<ClassName>::OnMyMouseEvent);

//You can write:
EventHandler<MouseEventArgs ^> ^handler += gcnew EvenHandler<MouseEventArgs
^>(this, &<ClassName>::OnMyMouseEvent);
```

The other four generic delegates found in the `System` namespace are designed to be used in conjunction with the static generic methods of `Array` or the `List<T>` type, but you can easily use them in other contexts:

[C#]

```
public delegate void Action<T>(T t);
public delegate int Comparison<T>(T x, T y);
public delegate U Converter<T, U>(T from);
public delegate bool Predicate<T>(T t);
```

[VB]

```
Public Delegate Sub Action(Of T)(ByVal t As T)
Public Delegate Function Comparison(Of T)(ByVal x As T, ByVal y As T) As Integer
Public Delegate Function Converter(Of T, U)(ByVal from As T) As U
Public Delegate Function Predicate(Of T)(ByVal t As T) As Boolean
```

[C++]

```
generic <typename T>
public delegate void Action(T t);
generic <typename T>
public delegate int Comparison(T x, T y);
generic <typename T, typename U>
public delegate U Converter(T from);
generic <typename T>
public delegate bool Predicate(T t);
```

For example, here is using the `Action<T>` delegate to trace every value in a given array:

[C#]

```
string[] cities = {"New York","San Francisico","London"};

Action<string> trace = delegate(string text)
                       {
                          Trace.WriteLine(text);
                       };
Array.ForEach(cities,trace);
```

[VB]

```
Sub TraceString(ByVal text As String)
   Trace.WriteLine(text)
End Sub

Dim cities() As String = {"New York", "San Francisico", "London"}
Dim actionDelegate As Action(Of String) = AddressOf TraceString

Array.ForEach(cities, actionDelegate)
```

[C++]

```
void TraceString(String ^text)
{
    Trace::WriteLine(text);
}

array <String ^> ^cities = {"New York","San Francisico","London"};

Action<String ^> ^trace = gcnew Action<String ^>(this, &<ClassName>::TraceString);
Array::ForEach(cities,trace);
```

What are the generic methods of System.Array?

The `System.Array` type is extended with many generic static methods. The generic static methods are designed to automate and streamline common tasks of working with arrays, such as iterating over the array and performing an action on each element, scanning the array looking for a value that matches a certain criteria (a predicate),

converting and sorting the array, and so on. Below is a partial listing of these static methods:

[C#]

```
public abstract class Array
{
   //Partial listing of the static methods:
   public static ReadOnlyCollection<T> AsReadOnly<T>(T[] array);
   public static int BinarySearch<T>(T[] array,T value);
   public static int BinarySearch<T>(T[] array,T value,
                                     IComparer<T> comparer);
   public static U[] ConvertAll<T,U>(T[] array,
                                Converter<T,U> converter);
   public static bool Exists<T>(T[] array,Predicate<T> match);
   public static T Find<T>(T[] array,Predicate<T> match);
   public static T[] FindAll<T>(T[] array,Predicate<T> match);
   public static int FindIndex<T>(T[] array,Predicate<T> match);
   public static void ForEach<T>(T[] array,Action<T> action);
   public static int  IndexOf<T>(T[] array,T value);
   public static void Sort<T>(T[] array,IComparer<T> comparer);
   public static void Sort<T>(T[] array,Comparison<T> comparison);
}
```

[VB]

```
Public MustInherit Class Array
    'Partial listing of the shared methods:
    Public Shared Function AsReadOnly(Of T)(ByVal array As T())
                                           As ReadOnlyCollection(Of T)
    Public Shared Function BinarySearch(Of T)(ByVal array As T(), _
                                             ByVal value As T) As Integer
    Public Shared Function BinarySearch(Of T)(ByVal array As T(), _
                                             ByVal value As T, _
                                             ByVal comparer As IComparer(Of T)) _
                                             As Integer
    Public Shared Function ConvertAll(Of T, U)(ByVal array As T(), _
                                              ByVal converter As
                                                    Converter(Of T, U)) As U()
    Public Shared Function Exists(Of T)(ByVal array As T(), _
                                       ByVal match As Predicate(Of T)) As Boolean
    Public Shared Function Find(Of T)(ByVal array As T(), _
                                     ByVal match As Predicate(Of T)) As T
    Public Shared Function FindAll(Of T)(ByVal array As T(), _
                                        ByVal match As Predicate(Of T)) As T()
    Public Shared Function FindIndex(Of T)(ByVal array As T(), _
                                          ByVal match As Predicate(Of T)) _
                                          As Integer
    Public Shared Sub ForEach(Of T)(ByVal array As T(), _
                                   ByVal action As Action(Of T))
    Public Shared Function IndexOf(Of T)(ByVal array As T(), ByVal value As T) _
                                        As Integer
    Public Shared Sub Sort(Of T)(ByVal array As T(),_
                                 ByVal comparer As IComparer(Of T))
    Public Shared Sub Sort(Of T)(ByVal array As T(), _
                                ByVal comparison As Comparison(Of T))
End Class
```

[C++]

```
public ref class Array abstract
{
   //Partial listing of the static methods:
public:
   generic <typename T>
   static ReadOnlyCollection<T> ^ AsReadOnly(array<T> ^arr);
   generic <typename T>
   static int BinarySearch (array<T> ^arr, T value);
   generic <typename T>
   static int BinarySearch (array<T> ^arr, T value,
                                    IComparer<T> ^comparer);
   generic <typename T, typename U>
   static array<U> ^ ConvertAll (array<T> ^arr,
                                 Converter<T,U> ^converter);
   generic <typename T>
   static bool Exists (array<T> ^arr,Predicate<T> ^match);
   generic <typename T>
   static T Find (array<T> ^arr,Predicate<T> ^match);
   generic <typename T>
   static array<T> ^ FindAll (array<T> ^arr, Predicate<T> ^match);
   generic <typename T>
   static int FindIndex (array<T> ^arr, Predicate<T> ^match);
   generic <typename T>
   static void ForEach (array<T> ^arr, Action<T> ^action);
   generic <typename T>
   static int  IndexOf (array<T> ^arr, T value);
   generic <typename T>
   static void Sort (array<T> ^arr,IComparer<T> ^comparer);
   generic <typename T>
   static void Sort (array<T> ^arr, Comparison<T> ^comparison) ;
};
```

Most of these static generic methods work with the four generic delegates defined in the `System` namespace:

[C#]

```
public delegate void Action<T>(T t);
public delegate int Comparison<T>(T x, T y);
public delegate U Converter<T, U>(T from);
public delegate bool Predicate<T>(T t);
```

[VB]

```
Public Delegate Sub Action(Of T)(ByVal t As T)
Public Delegate Function Comparison(Of T)(ByVal x As T, ByVal y As T) As Integer
Public Delegate Function Converter(Of T, U)(ByVal from As T) As U
Public Delegate Function Predicate(Of T)(ByVal t As T) As Boolean
```

[C++]

```
generic <typename T>
public delegate void Action(T t);
generic <typename T>
public delegate int Comparison(T x, T y);
generic <typename T, typename U>
public delegate U Converter(T from);
generic <typename T>
public delegate bool Predicate(T t);
```

For example, suppose the array `roles` contains all the roles a user plays at your application, and you would like to find out if the user is a member or a specified role.

[C#]

```
bool IsInRole(string role)
{
   string[] roles = GetRoles();

   Predicate<string> exists = delegate(string roleToMatch)
                              {
                                  return roleToMatch == role;
                              };
   return Array.Exists(roles,exists);
}
string[] GetRoles()
{...}
```

[VB]

```
Public Class SomeClass
   Dim m_RoleToMatch As String

   Private Function CompareRoles(ByVal role As String) As Boolean
      Return role = m_RoleToMatch
   End Function

   Public Function IsInRole(ByVal role As String) As Boolean
      Dim roles As String() = GetRoles()
      m_RoleToMatch = role
      Dim exists As Predicate(Of String)
      exists = New Predicate(Of String)(AddressOf CompareRoles)
      Return Array.Exists(roles, exists)
   End Function

   Private Function GetRoles() As String()
      ...
   End Function

End Class
```

[C++]

[Need C++ Code JL] please put code similar to the VB sample that also does not have anonymous methods.

The `Array.Exists()` method defined as:

[C#]

```
public static bool Exists<T>(T[] array,Predicate<T> match);
```

[VB]

```
Public Shared Function Exists(Of T)(ByVal array As T(), _
                                    ByVal match As Predicate(Of T)) As Boolean
```

[C++]

```
public: generic<typename T>
static bool Exists(array<T>^ array,Predicate<T>^ match);
```

takes a single type parameter (the type of the array). The compiler can infer the type automatically, so there is no need to specify that. The second parameter is a generic delegate of type `Predicate<T>()`, which returns a Boolean value. The `Array.Exists()` method iterates over the array, and invokes the predicate delegate on each item in the array. If the predicate returns `true`, it stops the iteration and returns `true`. If all the items in the array return `false` from invoking the predicate on them, `Array.Exists()` returns `false`. In C#, you can initialize the predicate using an anonymous method, and have `Array.Exists()` invoke that method on every item in the array until the predicate is satisfied or there are no more items.

To demystify how those various methods work, here is how `Array.Exist()` could be implemented:

[C#]

```
public abstract class Array
{
   public static bool Exists<T>(T[] array,Predicate<T> match)
   {
      if(array == null)
      {
         throw new ArgumentNullException("array");
      }
      if(match == null)
      {
         throw new ArgumentNullException("match");
      }
      foreach(T t in array)
      {
         if(match(t))
         {
            return true;
         }
      }
      return false;
   }
   //Rest of the methods
}
```

[VB]

```
Public MustInherit Class Array
    Public Shared Function Exists(Of T)(ByVal array As T(), _
                                        ByVal match As Predicate(Of T)) As Boolean
        If array Is Nothing Then Throw New ArgumentNullException("array")
        If match Is Nothing Then Throw New ArgumentNullException("match")
        For Each t As T In array
            If match(t) Then Return True
        Next t
        Return False
        ' Rest of the methods
    End Function
End Class
```

[C++]

[Need C++ Code JL]

What are the generic methods of List<T>?

Besides implementing `IList<T>`, the `List<T>` type contains many generic helper methods. These methods are designed to automate and streamline common tasks of working with the list, such as iterating over the list and performing a task on each element, scanning the list looking for a value that matches a certain criteria (a predicate), or just searching for a particular value, converting and sorting the list, and so on. Below is a partial listing of these generic methods:

[C#]

```
public class List<T> : IList<T>,
{
   //Partial listing of the generic helper methods:
   public List<U> ConvertAll<U>(Converter<T,U> converter);
   public bool Exists(Predicate<T> match);
   public T Find(Predicate<T> match);
   public List<T> FindAll(Predicate<T> match);
   public int FindIndex(Predicate<T> match);
   public T FindLast(Predicate<T> match);
   public void ForEach(Action<T> action);
   public int LastIndexOf(T item);
   public void Sort(Comparison<T> comparison);
   public T[] ToArray();
   //More members
}
```

[VB]

```
Public Class List(Of T)
    Implements IList(Of T)
    ' Partial listing of the generic helper methods:
    Public Function ConvertAll(Of U)(ByVal converter As Converter(Of T, U)) _
                                     As List(Of U)
    Public Function Exists(ByVal match As Predicate(Of T)) As Boolean
    Public Function Find(ByVal match As Predicate(Of T)) As T
    Public Function FindAll(ByVal match As Predicate(Of T)) As List(Of T)
    Public Function FindIndex(ByVal match As Predicate(Of T)) As Integer
    Public Function FindLast(ByVal match As Predicate(Of T)) As T
    Public Sub ForEach(ByVal action As Action(Of T))
    Public Function LastIndexOf(ByVal item As T) As Integer
    Public Sub Sort(ByVal comparison As Comparison(Of T))
    Public Function ToArray() As T()
    ' More members
End Class
```

[C++]

```
generic <typename T>
public ref class List : IList<T>, ICollection<T>, IEnumerable<T>,
                        IList, ICollection, IEnumerable
{
   //Partial listing of the geenric helper methods:
public:
   generic <typename T, typename U>
   List<U> ^ ConvertAll (Converter<T,U> ^converter);
   bool Exists(Predicate<T> ^match);
```

```
    T Find(Predicate<T> ^match);
    List<T> ^ FindAll(Predicate<T> ^match);
    int FindIndex(Predicate<T> ^match);
    T FindLast(Predicate<T> ^match);
    void ForEach(Action<T> ^action);
    int LastIndexOf(T item);
    void Sort(Comparison<T> ^comparison);
    array <T> ^ ToArray();
    //More members
};
```

Most of these helper generic methods work with the four generic delegates defined in the `System` namespace:

[C#]

```
public delegate void Action<T>(T t);
public delegate int Comparison<T>(T x, T y);
public delegate U Converter<T, U>(T from);
public delegate bool Predicate<T>(T t);
```

[VB]

```
Public Delegate Sub Action(Of T)(ByVal t As T)
Public Delegate Function Comparison(Of T)(ByVal x As T, ByVal y As T) As Integer
Public Delegate Function Converter(Of T, U)(ByVal from As T) As U
Public Delegate Function Predicate(Of T)(ByVal t As T) As Boolean
```

[C++]

```
generic <typename T>
public delegate void Action(T t);
generic <typename T>
public delegate int Comparison(T x, T y);
generic <typename T, typename U>
public delegate U Converter(T from);
generic <typename T>
public delegate bool Predicate(T t);
```

The List<T> helper methods are used much the same way as the generic static methods of `System.Array`. For example, the following code initializes a list with all the numbers from 1 to 20. Then, using the `Action<T>` delegate, the code traces these numbers using the `List<T>.ForEach()` method. Using the `Predicate<T>` delegate, the code finds all the prime numbers in the list by calling the `List<T>.FindAll()` method, which returns another list of the same type. Finally, the prime numbers are traced, using the same `Action<T>` delegate.

[C#]

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
List<int> list = new List<int>(numbers);

Action<int> trace =  delegate(int number)
                     {
                        Trace.WriteLine(number);
                     };
Predicate<int> isPrime =   delegate(int number)
                           {
                              switch(number)
```

```
                                {
                                   case 1:case 2:case 3:case 5:case 7:
                                   case 11:case 13:case 17:case 19:
                                      return true;
                                   default:
                                      return false;
                                }
                             };
list.ForEach(trace);
List<int> primes = list.FindAll(isPrime);
primes.ForEach(trace);
```

[VB]

```
Sub TraceNumber(ByVal number As Integer)
   Trace.WriteLine(number)
End Sub

Function IsPrimeNumber(ByVal number As Integer) As Boolean
   Select Case number
      Case 1,2,3,5,7,11,13,17,19
         Return True
      Case Else
         Return False
   End Select
End Function

Dim numbers() As Integer = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20}
   Dim list As New List(Of Integer)(numbers)

   Dim trace As Action(Of Integer)
   trace = New Action(Of Integer)(AddressOf TraceNumber)

   Dim isPrime = New Predicate(Of Integer)(AddressOf IsPrimeNumber)
   list.ForEach(trace)

   Dim primes As List(Of Integer) = list.FindAll(isPrime)
   primes.ForEach(trace)
End Sub
```

[C++]

```
Bool IsPrimeNumber(int number)
{
   switch(number)
   {
      case 1:case 2:case 3:case 5:case 7:
      case 11:case 13:case 17:case 19:
         return true;
      default:
         return false;
   }
}
void TraceNumber(int number)
{
   Trace::WriteLine(number);
}

int numbers[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
```

```
List<int> ^list = gcnew List<int>(numbers);

Action<int> ^trace = gcnew Action<int>(this,&<ClassName>::TraceNumber);
Predicate<int> ^isPrime =  gcnew Predicate<int>(this,&<ClassName>::IsPrimeNumber);

list->ForEach(trace);
List<int> ^primes = list->FindAll(isPrime);
primes->ForEach(trace);
```

What are nullable types?

Unlike reference types, you cannot assign a `null` into a value type. This is often a problem when interacting with code that interprets a `null` as having no value, rather than no-reference. The canonical example is database `null` values in columns that have representation as types such as `int` or `DateTime`. To address that, the `System` namespace provides the structure `Nullable<T>` defined as:

[C#]

```
public interface INullableValue
{
   bool HasValue{get;}
   object Value{get;}
}
[Serializable]
public struct Nullable<T> : INullableValue,IEquatable<Nullable<T>>,... where T :
struct
{
   public Nullable(T value);
   public bool HasValue{get;}
   public T Value{get;}
   public T GetValueOrDefault();
   public T GetValueOrDefault(T defaultValue);
   public bool Equals(Nullable<T> other);
   public static implicit operator Nullable<T>(T value);
   public static explicit operator T(Nullable<T> value);

   //More members
}
```

[VB]

```
Public Interface INullableValue
    ReadOnly Property HasValue() As Boolean
    ReadOnly Property Value() As Object
End Interface

Public Structure Nullable(Of T As Structure)
    Implements INullableValue,IEquatable(Of Nullable(Of T)), ' More interfaces

    Public Sub New(value As T)
    Public ReadOnly Property HasValue() As Boolean
    Public ReadOnly Property Value() As T
    Public Function GetValueOrDefault() As T
    Public Function GetValueOrDefault(ByVal defaultValue As T) As T
    Public Function Equals(ByVal other As Nullable(Of T)) As Boolean
    Public Shared Operator CType(ByVal value As T) As Nullable(Of T)
    Public Shared Operator CType(ByVal value As Nullable(Of T)) As T
```

```
    ' More members
End Structure
```

[C++]

```
public interface class INullableValue
{
   property bool HasValue{ bool get();}
   property Object ^ Value{ Object ^get();}
}
generic <typename T>
[Serializable]
public value struct Nullable : INullableValue,IEquatable<Nullable<T>>...
{
public:
   Nullable(T value);
   property bool HasValue { bool get(); }
   property T Value { T get(); }
   bool Equals(Nullable<T> other);
   T GetValueOrDefault();
   T GetValueOrDefault(T defaultValue);
   generic <typename T>
   static operator Nullable(T value);
   static explicit operator T(Nullable<T> value);
   //More members
};
```

Because the `Nullable<T>` struct uses a generic type parameter, you can use it to wrap a value type, and assign `null` into it:

[C#]

```
Nullable<int> number = 123;
Debug.Assert(number.HasValue);
number = null;
Debug.Assert(number.HasValue == false);
Debug.Assert(number.Equals(null));
```

[VB]

```
Dim number As Nullable(Of Integer) = 123
Debug.Assert(number.HasValue())
number = Nothing
Debug.Assert(number.HasValue() = False)
Debug.Assert(number.Equals(Nothing))
```

[C++]

```
Nullable<int> number = 123;
Debug::Assert(number.HasValue);
number = Nullable<int>::FromObject((Object ^)nullptr);
Debug::Assert(number.HasValue == false);
Debug::Assert(number.Equals(null));
```

Once a `null` is assigned to a nullable type, you can still access it to verify if it has a value, via the `HasValue` property, or just equate it to `null`.

In C# and VB, you can even use the underlying value type's operators on a nullable type:

[C#]

```
Nullable<int> number = 0;
number++;
```

[VB]

```
Dim number As Nullable(Of Integer) = 0
number += 1
```

The reason this is possible is because the compiler is capable of verifying that the underlying type supported the operator, and applying it on the value stored in the structure. This is called *lifted operators*.

The `Nullable<T>` struct also provides conversion operators, so you can convert a nullable type to and from a real value type:

[C#]

```
Nullable<int> nullableNumber = 123;
int number = (int)nullableNumber;
Debug.Assert(number == 123);

number = 456;
nullableNumber = number;
Debug.Assert(nullableNumber.Equals(456));
```

[VB]

```
Dim nullableNumber As Nullable(Of Integer) = 123
Dim number As Integer = CType(nullableNumber, Integer)
Debug.Assert(number = 123)

number = 456
nullableNumber = number
Debug.Assert(nullableNumber.Equals(456))
```

[C++]

```
Nullable<int> nullableNumber = 123;
int number = (int)nullableNumber
Debug::Assert(number == 123);

number = 456;
nullableNumber = number;
Debug::Assert(nullableNumber.Equals(456));
```

Note that using `Nullable<T>` on `Nullable<T>` is disallowed, and the compiler will issue an error:

[C#]

```
//This will not compile:
Nullable<Nullable<int>> number = 123;
```

[VB]

```
' This will not compile:
Dim number As Nullable(Of Nullable(Of Integer)) = 123
```

[C++]

```
//This will not compile:
Nullable<Nullable<int>> number = 123;
```

You can use the overloaded methods `GetValueOrDefault()` of `Nullable<T>` to defensively obtain either the value stored in the nullable type or it its default, if it does contain a `null`:

[C#]

```
Nullable<DateTime> time = null;
DateTime value = time.GetValueOrDefault();
Debug.Assert(value.ToString() == "1/1/0001 12:00:00 AM");
```

[VB]

```
Dim time As Nullable(Of DateTime)
Dim value As DateTime = time.GetValueOrDefault()
Debug.Assert(value.ToString() = "1/1/0001 12:00:00 AM")
```

[C++]

```
Nullable<DateTime> time = null;
DateTime value = time.GetValueOrDefault();
Debug.Assert(value.ToString() == "1/1/0001 12:00:00 AM");
```

The `System` namespace also defines the static helper class `Nullable` and the helper class `NullableConverter`, but those are not needed usually.

The C# 2.0 compiler supports shorthand for `Nullable<T>`. You can use the `?` modifier on value types to actually construct a `Nullable<T>` around it:

```
int? number = 123;
Debug.Assert(number.HasValue);
number = null;
Debug.Assert(number.HasValue == false);
```

Note that the type declared by the `?` modifier is identical to that created using `Nullable<T>` directly:

```
Debug.Assert(typeof(int?) == typeof(Nullable<int>));
```

As with using `Nullable<T>` directly, the compiler supports lifted operators. Whenever you combine nullable types using operators, if any one of them is `null`, then the resulting expression will be `null` too:

```
int? number1 = 123;
int? number2 = null;
int? sum = number1 + number2;
Debug.Assert(sum == null);
```

Using the `?` modifier is the common way of declaring and using nullable variables in C#. You can even pass nullable types as type arguments for generic types:

```
IList<int?> list = new List<int?>();
list.Add(3);
list.Add(null);
```

C# 2.0 also provides the *null coalescing operator* via the `??` operator.

```
c = a ?? b;
```

The result of applying the ?? operator on two operands returns the left-hand side operand (`a`) if it is not `null`, and the right operand (`b`)otherwise. While `b` can of course be `null` too, you typically use the `??` operator to supply a default value in case `a` is null.

How do I reflect generic types?

Like most other things done with reflection, you use the class `Type`. `Type` can represent generic types with specific type arguments (called *bounded types*), or unspecified (unbounded) types.

[C#]

Both `typeof` and `GetType()` can operate on type parameters:

```
public class MyClass<T>
{
   public void SomeMethod(T t)
   {
      Type type = typeof(T);
      Debug.Assert(type == t.GetType());
   }
}
```

In addition the `typeof` operator can operate on *unbound generic types* (generic types that do not have yet specific type arguments). For example:

```
public class MyClass<T>
{}
Type unboundedType = typeof(MyClass<>);
Trace.WriteLine(unboundedType.ToString());
//Writes: MyClass`1[T]
```

The number `1` being traced is the number of generic type parameters of the generic type used. Note the use of the empty <>. To operate on an unbound generic type with multiple type parameters, use a `,` in the <>:

```
public class LinkedList<K,T>
{...}
Type unboundedList = typeof(LinkedList<,>);
Trace.WriteLine(unboundedList.ToString());
//Writes: LinkedList`2[K,T]
```

[VB]

Both `GetType()` and `Object.GetType()` can operate on type parameters:

```
Public Class SomeClass(Of T)
   Public Sub SomeMethod(ByVal t As T)
      Dim theType As Type = GetType(T)
      Debug.Assert((theType Is t.GetType))
   End Sub
End Class
```

[C++]

Both `typeid<>` and `GetType()` can operate on type parameters:

```
generic <typename T>
public ref class MyClass
{
public:
   void SomeMethod(T t)
   {
      Type ^type = typeid<T>;
```

```
        Debug::Assert(type == t->GetType());
    }
};
```

To support generics, `Type` has special methods and properties designed to provide reflection information about the generic aspects of the type:

[C#]

```
public abstract class Type : //Base types
{
   public virtual bool ContainsGenericParameters{get;}
   public virtual GenericParameterAttributes GenericParameterAttributes{get;}
   public virtual int GenericParameterPosition{get;}
   public virtual bool IsGenericType{get;}
   public virtual bool IsGenericParameter{get;}
   public virtual bool IsGenericTypeDefinition{get;}
   public virtual Type[] GetGenericArguments();
   public virtual Type[] GetGenericParameterConstraints();
   public virtual Type GetGenericTypeDefinition();
   public virtual Type MakeGenericType(params Type[] typeArguments);
   //Rest of the members
}
```

[VB]

```
Public MustInherit Class Type ' Base types
   Public Overridable ReadOnly Property ContainsGenericParameters As Boolean
   Public Overridable ReadOnly Property GenericParameterAttributes As
                                            GenericParameterAttributes
   Public Overridable ReadOnly Property GenericParameterPosition As Integer
   Public Overridable ReadOnly Property IsGenericType As Boolean
   Public Overridable ReadOnly Property IsGenericParameter As Boolean
   Public Overridable ReadOnly Property IsGenericTypeDefinition As Boolean
   Public Overridable Function GetGenericArguments() As Type()
   Public Overridable Function GetGenericParameterConstraints() As Type()
   Public Overridable Function GetGenericTypeDefinition() As Type
   Public Overridable Function MakeGenericType(ByVal ParamArray typeArguments As
                                                    Type()) As Type
   ' Rest of the members
End Class
```

[C++]

```
public ref class Type abstract : //Base types
{
public:
   property virtual GenericParameterAttributes GenericParameterAttributes{
                                       GenericParameterAttributes get;}
   property virtual bool ContainsGenericParameters{ bool get();}
   property virtual int GenericParameterPosition{ int get();}
   property virtual bool IsGenericType{ bool get();}
   property virtual bool IsGenericParameter{bool get();}
   property virtual bool IsGenericTypeDefinition{ bool get();}
   virtual array<Type ^> ^ GetGenericArguments();
   virtual array<Type^>^ GetGenericParameterConstraints();
   virtual Type ^ GetGenericTypeDefinition();
   virtual Type^ MakeGenericType(... array<Type^>^ typeArguments);
```

```
    //Rest of the members
};
```

The most useful of these new members are the `IsGenericType` property, the `GetGenericArguments()` and `GetGenericTypeDefinition()` methods. As its name indicates, `IsGenericType` is set to `true` if the type represented by the `Type` object uses generic type parameters. `GetGenericArguments()` returns an array of types corresponding to the type arguments used. `GetGenericTypeDefinition()` returns a `Type` representing the generic form of the underlying type. The following example demonstrates using these generic-handling `Type` members to obtain generic reflection information on a generic linked list.

[C#]

```
public class LinkedList<K,T>
{...}

LinkedList<int,string> list = new LinkedList<int,string>();

Type boundedType = list.GetType();
Trace.WriteLine(boundedType.ToString());
//Writes: LinkedList`2[System.Int32,System.String]

Debug.Assert(boundedType.IsGenericType);

Type[] parameters = boundedType.GetGenericArguments();

Debug.Assert(parameters.Length == 2);
Debug.Assert(parameters[0] == typeof(int));
Debug.Assert(parameters[1] == typeof(string));

Type unboundedType = boundedType.GetGenericTypeDefinition();
Trace.WriteLine(unboundedType.ToString());
//Writes: LinkedList`2[K,T]
```

[VB]

```
Class LinkedList(Of T, K)
    ...
End Class

Dim list As New LinkedList(Of Integer, String)
Dim listType As Type = list.GetType()
Trace.WriteLine(listType.ToString)
' Writes: LinkedList`2[System.Int32,System.String]

Debug.Assert(listType.IsGenericType)

Dim parameters As Type() = listType.GetGenericArguments()

Debug.Assert(parameters.Length = 2)
Debug.Assert(parameters(0) Is GetType(Integer))
Debug.Assert(parameters(1) Is GetType(String))

Dim unboundedType As Type = listType.GetGenericTypeDefinition()
Trace.WriteLine(unboundedType.ToString)
' Writes: LinkedList`2[K,T]
```

[C++]

```
generic <typename K, typename T>
public ref class LinkedList
{...};

LinkedList<int,String ^> ^list = gcnew LinkedList<int,String ^>;

Type ^boundedType = list->GetType();
Trace::WriteLine(boundedType->ToString());
//Writes: LinkedList`2[System.Int32,System.String]

Debug::Assert(boundedType->IsGenericType);

array <Type ^> ^parameters = boundedType->GetGenericArguments();

Debug::Assert(parameters->Length == 2);
Debug::Assert(parameters[0] == typeid<int>);
Debug::Assert(parameters[1] == typeid<String ^>);

Type ^unboundedType = boundedType->GetGenericTypeDefinition();
Trace::WriteLine(unboundedType->ToString());
//Writes: LinkedList`2[K,T]
```

Tools Support

How does Visual Studio 2005 support generics?

Visual Studio 2005 supports generics well. InteliSense displays correctly the generic types, implementing generic interfaces is just as easy as with non-generic interfaces. The most impressive aspect of support is in the debugger, which displays the correct type arguments information when hovering over generic types.

Can I data-bind generic types to Windows and Web data controls?

Yes. All the generic collections also support the non-generic collection interfaces, and you can use them as data sources to bind to controls just as with the non-generics collections.

For example, consider a Windows Forms form that has a combobox called `m_ComboBox`. You can assign into as a data source the `List<T>` collection:

[C#]

```
partial class MyForm : Form
{
   void OnFormLoad(object sender, EventArgs e)
   {
      List<string> cities = new List<string>();
      cities.Add("New York");
      cities.Add("San Francisico");
      cities.Add("London");

      m_ComboBox.DataSource = cities;
   }
}
```

[VB]

```
Public Class MyForm
   Inherits Form
   Private Sub OnFormLoad(ByVal sender As System.Object,
                          ByVal e As System.EventArgs) Handles MyBase.Load

      Dim cities As New List(Of String)
      cities.Add("New York")
      cities.Add("San Francisico")
      cities.Add("London")
      m_ComboBox.DataSource = cities
   End Sub
End Class
```

[C++]

```
public ref class MyForm : public Form
{
   void Form_Load(Object^ sender,EventArgs^ e)
   {
      List<String ^> ^cities = gcnew List<String ^>;
      cities->Add("New York");
      cities->Add("San Francisico");
      cities->Add("London");
      m_ComboBox->DataSource = cities;
   }
};
```

How are Web Service proxies created for generic types?

The web service proxy class generated by Visual Studio 2005 does not necessarily maintain affinity to the returned types from a web service. The proxy class will contain values corresponding only to the serialized representation of the generic types only.

As mentioned in the question on generics and web services, for this definition of a web service:

[C#]

```
public class MyWebService
{
   [WebMethod]
   public List<string> GetCities()
   {
```

```
      List<string> cities = new List<string>();
      cities.Add("New York");
      cities.Add("San Francisco");
      cities.Add("London");
      return cities;
   }
}
```

[VB]

```
Public Class MyWebService
    [WebMethod]
    Public Function GetCities() As List(Of String)
        Dim cities As New List(Of String)()
        cities.add("New York")
        Cities.add("San Francisco")
        cities.add("London")
        Return cities
    End Function
End Class
```

[C++]

```
public ref class MyWebService
{
   public:
   [WebMethod]
   List<String ^> ^ GetCities()
   {
      List<String ^> ^cities = gcnew List<String ^>();
      cities->Add("New York");
      cities->Add("San Francisco");
      cities->Add("London");
      return cities;
   }
}
```

The returned list will be marshaled as an array of strings. Consequently, the Visual Studio 2005 generated proxy will contain this definition of the GetCities() method:

[C#]

```
[WebServiceBinding(Name="MyWebServiceSoap")]
public partial class MyWebService : SoapHttpClientProtocol
{
   public MyWebService()
   {...}

   [SoapDocumentMethod(...)]
   public string[] GetCities()
   {
      object[] results = Invoke("GetCities",new object[]{});
      return ((string[])(results[0]));
   }
}
```

[VB]

```
<WebServiceBinding(Name:="MyWebServiceSoap")> _
```

```
Public Partial Class MyWebService
      Inherits SoapHttpClientProtocol

   Public Sub New()
      ...
   End Sub

   <SoapDocumentMethod(...)> _
   Public Function GetCities() As String()
      Dim results As Object() = Invoke("GetCities", New Object(){})
      Return CType(results(0), String())
   End Function
End Class
```

[C++]

```
[WebServiceBinding(Name=L"MyWebServiceSoap")]
public ref class MyWebService : public SoapHttpClientProtocol
{
   public: Service::Service()
   {...}

   public:[SoapDocumentMethod(...)]
   cli::array<String^  >^ GetCities()
   {
      cli::array<Object^  >^ results = Invoke(L"GetCities",
                              gcnew cli::array<Object^>(0));
      return (cli::safe_cast<cli::array< System::String^>^>(results[0]);
    }
}
```

Best Practices

When should I not use generics?

The main reason not to use generics is cross-targeting – if you build the same code for both .NET 1.1 and .NET 2.0, then you cannot take advantage of generics, since they are only supported on .NET 2.0.

What naming convention should I use for generics?

I recommend using a single capital letter for a generic type parameter. If you have no additional contextual information about the type parameter, you should use the letter `T`:

[C#]

```
public class MyClass<T>
{...}
```

[VB]

```
Public Class SomeClass(Of T)
   ...
End Class
```

[C++]

```
generic <typename T>
public ref class MyClass
```

```
{...};
```

In all other cases, the official Microsoft guidelines for generic naming conventions are:

- Name generic type parameters with descriptive names, unless a single letter name is completely self explanatory and a descriptive name would not add value.

  [C#]

```
public interface ISessionChannel<TSession>
{...}
public delegate TOutput Converter<TInput,TOutput>(TInput from);
```

  [VB]

```
Public Interface ISessionChannel(Of TSession)
   ...
End Interface

Public Delegate Function Converter(Of TInput, TOutput)(ByVal input As TInput)
                                                                    As TOutput
```

  [C++]

```
generic <typename TSession>
public interface class ISessionChannel
{...};
generic <typename TInput, typename TOutput>
public delegate TOutput Converter(TInput from);
```

- Consider indicating constraints placed on a type parameter in the name of parameter. For example, a parameter constrained to `ISession` may be called `TSession`.

Should I put constraints on generic interfaces?

An interface can define constraints for the generic types it uses. For example,

[C#]

```
public interface ILinkedList<T> where T : IComparable<T>
{...}
```

[VB]

```
Public Interface ILinkedList(Of T As IComparable(Of T))
   ...
End Interface
```

[C++]

```
generic <typename T> where T : IComparable<T>
public interface class ILinkedList
{...};
```

However, you should be very mindful about the implications of defining constraints at the scope of an interface. An interface should not have any shred of implementation details, to reinforce the notion of separation of interface from implementation. There are many ways in which one could implement the generic interface. The specific type arguments used are, after all, an implementation detail. Constraining them commonly couples the interface to specific implementation options.

It is better to let the class implementing the generic interface add the constraint and keep the interface itself constraints-free:

[C#]

```
public class LinkedList<T> : ILinkedList<T> where T : IComparable<T>
{
   //Rest of the implementation
}
```

[VB]

```
Public Class LinkedList(Of T As IComparable(Of T))
       Implements ILinkedList(Of T)
' Rest of the implementation
End Class
```

[C++]

```
generic <typename T> where T : IComparable<T>
public ref class LinkedList : ILinkedList<T>
{
   //Rest of the implementation
};
```

How do I dispose of a generic type?

In C# and VB, when you supply an object of a generic type parameter to the `using` statement, the compiler has no way of knowing whether the actual type the client will specify supports `IDisposable`. The compiler will therefore not allow you to specify an instance of a generic type parameter for the using statement:

[C#]

```
public class MyClass<T>
{
   public void SomeMethod(T t)
   {
      using(t)//Does not compile
      {...}
   }
}
```

[VB]

```
Public Class SomeClass(Of T)
   Public Sub SomeMethod(ByVal value As T)
      Using value ' Does not compile
      End Using
   End Sub
End Class
```

Instead, you can constrain the type parameter to support `IDisposable`:

[C#]

```
public class MyClass<T> where T : IDisposable
{
   public void SomeMethod(T t)
   {
      using(t)
```

```
      {...}
   }
}
```

[VB]

```
Public Class SomeClass(Of T As IDisposable)
   Public Sub SomeMethod(ByVal value As T)
      Using value
      End Using
   End Sub
End Class
```

However, you should not do so. The problem with the `IDisposable` constraint is that now you cannot use interfaces as type arguments, even if the underlying type supports `IDisposable`:

[C#]

```
public interface IMyInterface
{}
public class MyOtherClass : IMyInterface,IDisposable
{...}
public class MyClass<T> where T : IDisposable
{
   public void SomeMethod(T t)
   {
      using(t)
      {...}
   }
}
MyOtherClass myOtherClass = new MyOtherClass();
MyClass<IMyInterface> obj = new MyClass<IMyInterface>();//Does not compile
obj.SomeMethod(myOtherClass);
```

[VB]

```
Public Interface IMyInterface
End Interface

Public Class MyOtherClass
      Implements IMyInterface, IDisposable
   ...
End Class

Public Class SomeClass(Of T As IDisposable)
   Public Sub SomeMethod(ByVal value As T)
      Using value
      End Using
   End Sub
End Class

Dim myOtherClass As New MyOtherClass
Dim obj As New SomeClass(Of IMyInterface) ' Does not compile
obj.SomeMethod(myOtherClass)
```

Instead of constraining the type parameter to derive from IDisposable, I recommend that you use the as operator in C# or the TryCast operator in VB with the using statement on generic type parameters to enable its use when dealing with interfaces:

[C#]

```
public class MyClass<T>
{
   public void SomeMethod(T t)
   {
      using(t as IDisposable)
      {...}
   }
}
```

[VB]

```
Public Class SomeClass(Of T)
   Public Sub SomeMethod(ByVal value As T)
      Using TryCast(value, IDisposable)
      End Using
   End Sub
End Class
```

Can I cast to and from generic type parameters?

The compiler will only let you implicitly cast generic type parameters to `object`, or to constraint-specified types:

[C#]

```
interface ISomeInterface
{...}
class BaseClass
{...}
class MyClass<T> where T : BaseClass,ISomeInterface
{
   void SomeMethod(T t)
   {
      ISomeInterface obj1 = t;
      BaseClass      obj2 = t;
      object         obj3 = t;
   }
}
```

[VB]

```
Interface ISomeInterface
   ...
End Interface

Class BaseClass
   ...
End Class

Class SomeClass(Of T As{BaseClass,ISomeInterface})

   Private Sub SomeMethod(ByVal value As T)
      Dim obj1 As ISomeInterface = value
      Dim obj2 As BaseClass = value
      Dim obj3 As Object = value
   End Sub
End Class
```

[C++]

```
interface class ISomeInterface
{...};
ref class BaseClass
{...};
generic <typename T> where T : BaseClass,ISomeInterface
ref class MyClass
{
   void SomeMethod(T t)
   {
      ISomeInterface ^obj1 = t;
      BaseClass      ^obj2 = t;
      Object         ^obj3 = t;
   }
};
```

Such implicit casting is of course type safe, because any incompatibility is discovered at compile-time.

The compiler will let you explicitly cast generic type parameters to any interface, but not to a class:

[C#]

```
interface ISomeInterface
{...}
class SomeClass
{...}
class MyClass<T>
{
   void SomeMethod(T t)
   {
      ISomeInterface obj1 = (ISomeInterface)t;//Compiles
      SomeClass      obj2 = (SomeClass)t;     //Does not compile
   }
}
```

[VB]

```
Interface ISomeInterface
   ...
End Interface

Class BaseClass
   ...
End Class

Class SomeClass(Of T)
   Private Sub SomeMethod(ByVal value As T)
      Dim obj1 As ISomeInterface = CType(value,ISomeInterface)' Compiles
      Dim obj2 As BaseClass = CType(value,BaseClass)' Does not compile
   End Sub
End Class
```

[C++]

```
interface class ISomeInterface
{...};
ref class SomeClass
```

```
{...};
generic <typename T>
ref class MyClass
{
   void SomeMethod(T t)
   {
      ISomeInterface ^obj1 = (ISomeInterface ^)t;//Compiles
      SomeClass      ^obj2 = (SomeClass ^)t;     //Does not compile
   }
};
```

However, you can force a cast from a generic type parameter to any other type using a temporary `object` variable:

[C#]

```
class MyOtherClass
{...}

class MyClass<T>
{
   void SomeMethod(T t)
   {
      object temp = t;
      MyOtherClass obj = (MyOtherClass)temp;
   }
}
```

[VB]

```
Class MyOtherClass
   ...
End Class

Class SomeClass(Of T)
   Sub SomeMethod(ByVal value As T)
      Dim temp As Object = value
      Dim obj As MyOtherClass = CType(temp, MyOtherClass)
   End Sub
End Class
```

[C++]

```
ref class SomeClass
{...};

generic <typename T>
ref class MyClass
{
   void SomeMethod(T t)
   {
      Object ^temp = t;
      SomeClass ^obj = (SomeClass ^)temp;
   }
};
```

Needless to say, such explicit casting is dangerous because it may throw an exception at runtime if the concrete type used instead of the generic type parameter does not derive from the type you explicitly cast to.

[C#]

Instead of risking a casting exception, a better approach is to use the `is` or `as` operators. The `is` operator returns `true` if the generic type parameter is of the queried type, and `as` will perform a cast if the types are compatible, and will return `null` otherwise.

```
public class MyClass<T>
{
   public void SomeMethod(T t)
   {
      if(t is int)
      {...}

      if(t is LinkedList<int,string>)
      {...}

      string str = t as string;
      if(str != null)
      {...}

      LinkedList<int,string> list = t as LinkedList<int,string>;
      if(list != null)
      {...}
   }
}
```

[VB]

Instead of risking a casting exception, a better approach is to use the `TypeOf` and the `TryCast` operators. The `is` operator returns `true` if the generic type parameter is of the queried type. You can also use the `TryCast` operator to try to perform a cast if the types are compatible, and return `Nothing` otherwise.

```
Class SomeClass(Of T)

   Public Sub SomeMethod(ByVal value As T)
      If TypeOf value Is Integer Then
         ...
      End If

      If TypeOf value Is LinkedList(Of Integer, String) Then
         ...
      End If

      Dim str As String = TryCast(value,String)
      If (Not str Is Nothing) Then
         ...
      End If

      Dim list As LinkedList(Of Integer, String) = TryCast(value,LinkedList(Of
                                                   Integer, String))
      If (Not list Is Nothing) Then
         ...
      End If
   End Sub
End Class
```

How do I synchronize multithreaded access to a generic type?

In general, you should not use a `Monitor` on generic type parameters. The reason is that the `Monitor` can only be used with reference types. When you use generic types, the compiler cannot tell in advance whether you will provide a reference or a value type parameter. In C#, the compiler will let you use the `lock()` statement, yet if you provide a value type as the type parameter, it will have no effect at runtime. In VB, the compiler will not let you use the SyncLock on generic type parameters if the compiler is not certain the generic type parameter is a reference type.

In C# and VB, the only time when you could safely lock the generic type parameter is when you can constrain it to be a reference type, either by constraining it to be a reference type, or to derive from a base class:

[C#]

```
public class MyClass<T> where T : class
{..}
```

[VB]

```
Public Class SomeClass(Of T As Class)
   ...
End Class
```

or:

[C#]

```
public class SomeClass
{...}
public class MyClass<T> where T : SomeClass
{...}
```

[VB]

```
Public Class SomeClass
   ...
End Class
Public Class SomeClass(Of T As SomeClass)
   ...
End Class
```

Yet in general with synchronization it is better to avoid fragmented locking of individual member variables because that raises the likelihood of deadlocks.

How do I serialize generic types?

A generic class that has generic type parameters as members can be marked for serialization:

[C#]

```
[Serializable]
public class MySerializableClass<T>
{
   T m_T;
}
```

[VB]

```
<Serializable()> _
Public Class MySerializableClass(Of T)

   Dim m_T As T
End Class
```

[C++]

```
generic <typename T>
[Serializable]
public ref class MyClass
{
   T m_T;
};
```

However, in such cases, the generic class is only serializable if the generic type parameter specified is serializable. Consider this code:

[C#]

```
public class SomeClass
{}
MySerializableClass<SomeClass> obj;
```

[VB]

```
Public Class SomeClass
End Class

Dim obj as MySerializableClass(Of SomeClass)
```

[C++]

```
public ref class SomeClass
{};
MyClass<SomeClass ^> ^obj;
```

`obj` is not serializable because the type parameter `SomeClass` is not serializable. Consequently, `MySerializableClass<T>` may or may not be serializable, depending on the generic type parameter used. This may result in a run-time loss of data or system corruption, because the client application may not be able to persist the state of the object.

Presently, .NET does not provide a mechanism for constraining a generic type parameter to be serializable. The workaround is to perform a single run-time check before any use of the type, and abort the use immediately, before any damage could take place. You can place the run-time verification in the static constructor:

[C#]

```
[Serializable]
class MySerializableClass<T>
{
   T m_T;

   static MySerializableClass()
   {
      ConstrainType(typeof(T));
   }
   static void ConstrainType(Type type)
```

```
   {
      bool serializable = type.IsSerializable;
      if(serializable == false)
      {
         string message = "The type " + type + " is not serializable";
         throw new InvalidOperationException(message);
      }
   }
}
```

[VB]

```
<Serializable()> _
Class SomeClass(Of T)
   Private m_T As T

   Shared Sub New()
      ConstrainType(GetType(T))
   End Sub
   Private Shared Sub ConstrainType(ByVal t As Type)
      If Not t.IsSerializable Then
         Dim message As String = "The type " + t.ToString() + " is not
                                                              serializable"
         Throw New InvalidOperationException(message)
      End If
   End Sub
End Class
```

[C++]

```
generic <typename T>
[Serializable]
ref class MyClass
{
   T m_T;
public:
   static MyClass()
   {
      ConstrainType(typeid<T>);
   }
private:
   static void ConstrainType(Type type)
   {
      bool serializable = type->IsSerializable;
      if(serializable == false)
      {
         String ^message = String::Concat("The type ", type->Name, " is not
serializable");
         throw gcnew SerializationException(message);
      }
   }
};
```

The static constructor is invoked exactly once per type per app domain, upon the first attempt to instantiate an object of that type. Performing the constraint verification in the static constructor is a technique applicable to any constraint that you cannot enforce at compile time, yet you have some programmatic way of determining and enforcing it at runtime.

About Juval Lowy

Juval Lowy is a software architect and the principal of IDesign, specializing in .NET architecture consulting and advanced .NET training. Juval is Microsoft's Regional Director for the Silicon Valley, working with Microsoft on helping the industry adopt .NET. His latest book is Programming .NET Components 2nd Edition (O'Reilly, 2005). Juval participates in the Microsoft internal design reviews for future versions of .NET. Juval published numerous articles, regarding almost every aspect of .NET development, and is a frequent presenter at development conferences. Microsoft recognized Juval as a Software Legend as one of the world's top .NET experts and industry leaders.