Search MSDN with Bing

MSDN Library
Development Tools and Languages
Visual Studio 2005
Technical Articles
C#
Creating a Debugger Visualizer Using Vis
ExpressPaint
An Extensive Examination of Data Structur
An Introduction to C# Generics
An Introduction to Programming Using M
Investigating Code Snippet Technology
An Overview of Web Browser Express
Refactoring C# Code Using Visual Studio
A Sneak Preview of Visual C# Whidbey
Using the Edit and Continue Feature in C#
Using Settings in C#
Working with the C# 2.0 Command Line C

An Introduction to C# Generics

msdn

513 out of 681 rated this helpful Rate this topic

Visual Studio 2005

Juval Lowy
IDesign

Updated January 2005

Summary: This article discusses the problem space generics address, how they are implemented, the benefits of the programming model, and unique innovations, such as constrains, generic methods and delegates, and generic inheritance. In addition it discusses how the .NET Framework utilizes generics (45 pages)

MSDNsamples/GenericsInCSharp

Download the GenericsInCSharp.msi sample file.

> **Note** This article assumes you are familiar with C# 1.1. For more information on the C# language, visit http://msdn.microsoft.com/vcsharp/language.

Contents

Introduction

Generics are the most powerful feature of C# 2.0. Generics allow you to define type-safe data structures, without committing to actual data types. This results in a significant performance boost and higher quality code, because you get to reuse data processing algorithms without duplicating type-specific code. In concept, generics are similar to C++ templates, but are drastically different in implementation and capabilities. This article discusses the problem space generics address, how they are implemented, the benefits of the programming model, and unique innovations, such as constrains, generic methods and delegates, and generic inheritance. You will also see how generics are utilized in other areas of the .NET Framework such as reflection, arrays, collections, serialization, and remoting, and how to improve on the basic offering.

Generics Problem Statement

Consider an everyday data structure such as a stack, providing the classic **Push()** and **Pop()** methods. When developing a general-purpose stack, you would like to use it to store instances of various types. Under C# 1.1, you have to use an Object-based stack, meaning that the internal data type used in the stack is an amorphous Object, and the stack methods interact with Objects:

```
public class Stack
{
   object[] m_Items;
   public void Push(object item)
   {...}
   public object Pop()
   {...}
}
```

Code block 1 shows the full implementation of the Object-based stack. Because Object is the canonical .NET base type, you can use the Object-based stack to hold any type of items, such as integers:

```
Stack stack = new Stack();
stack.Push(1);
stack.Push(2);
int number = (int)stack.Pop();
```

Code block 1. An Object-based stack

```
public class Stack
{
   readonly int m_Size;
   int m_StackPointer = 0;
   object[] m_Items;
   public Stack():this(100)
   {}
   public Stack(int size)
   {
      m_Size = size;
      m_Items = new object[m_Size];
   }
   public void Push(object item)
   {
      if(m_StackPointer >= m_Size)
         throw new StackOverflowException();
      m_Items[m_StackPointer] = item;
      m_StackPointer++;
   }
   public object Pop()
   {
      m_StackPointer--;
      if(m_StackPointer >= 0)
      {
         return m_Items[m_StackPointer];
      }
      else
      {
         m_StackPointer = 0;
         throw new InvalidOperationException("Cannot pop an empty stack");
      }
   }
}
```

However, there are two problems with Object-based solutions. The first issue is performance. When using value types, you have to box them in order to push and store them, and unbox the value types when popping them off the stack. Boxing and unboxing incurs a significant performance penalty in their own right, but it also increases the pressure on the managed heap, resulting in more garbage collections, which is not great for performance either. Even when using reference types instead of value types, there is still a performance penalty because you have to cast from an Object to the actual type you interact with and incur the casting cost:

```
Stack stack = new Stack();
stack.Push("1");
string number = (string)stack.Pop();
```

The second (and often more severe) problem with the Object-based solution is type safety. Because the compiler lets you cast anything to and from Object, you lose compile-time type safety. For example, the following code compiles fine, but raises an invalid cast exception at run time:

```
Stack stack = new Stack();
stack.Push(1);
//This compiles, but is not type safe, and will throw an exception:
string number = (string)stack.Pop();
```

You can overcome these two problems by providing a type-specific (and hence, type-safe) performant stack. For integers you can implement and use the **IntStack**:

```
public class IntStack
{
   int[] m_Items;
   public void Push(int item){...}
   public int Pop(){...}
}
IntStack stack = new IntStack();
stack.Push(1);
int number = stack.Pop();
```

For strings you would implement the **StringStack**:

```
public class StringStack
{
   string[] m_Items;
   public void Push(string item){...}
   public string Pop(){...}
}
StringStack stack = new StringStack();
stack.Push("1");
string number = stack.Pop();
```

And so on. Unfortunately, solving the performance and type-safety problems this way introduces a third, and just as serious problem—productivity impact. Writing type-specific data structures is a tedious, repetitive, and error-prone task. When fixing 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 is essentially the same data structure. In addition, there is no way to foresee the use of unknown or yet-undefined future types, so you have to keep an Object-based data structure as well. As a result, most C# 1.1 developers found type-specific data structures to be impractical and opt for using Object-based data structures, in spite of their deficiencies.

What Are Generics

Generics allow you to define type-safe classes without compromising type safety, performance, or productivity. You implement the server only once as a generic server, while at the same time you can declare and use it with any type. To do that, use the **<** and **>** brackets, enclosing a generic type parameter. For example, here is how you define and use a generic stack:

```
public class Stack<T>
{
   T[] m_Items;
   public void Push(T item)
   {...}
   public T Pop()
   {...}
}
Stack<int> stack = new Stack<int>();
stack.Push(1);
stack.Push(2);
int number = stack.Pop();
```

Code block 2 shows the full implementation of the generic stack. Compare **Code block 1** to **Code block 2** and see that it is as if every use of `object` in **Code block 1** is replaced with `T` in **Code block 2**, except that the Stack is defined using the generic type parameter T:

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

When using a generic stack, you have to instruct the compiler which type to use instead of the generic type parameter T, both when declaring the variable and when instantiating it:

```
Stack<int> stack = new Stack<int>();
```

The compiler and the runtime do the rest. All the methods (or properties) that accept or return a T will instead use the specified type, an integer in the example above.

Code block 2. The generic stack

```
public class Stack<T>
{
   readonly int m_Size;
   int m_StackPointer = 0;
   T[] m_Items;
   public Stack():this(100)
   {}
   public Stack(int size)
   {
      m_Size = size;
      m_Items = new T[m_Size];
   }
   public void Push(T item)
   {
      if(m_StackPointer >= m_Size)
         throw new StackOverflowException();
      m_Items[m_StackPointer] = item;
      m_StackPointer++;
   }
   public T Pop()
   {
      m_StackPointer--;
      if(m_StackPointer >= 0)
      {
         return m_Items[m_StackPointer];
      }
      else
      {
         m_StackPointer = 0;
         throw new InvalidOperationException("Cannot pop an empty stack");
      }
   }
}
```

> **Note** T is the generic type parameter (or type parameter) while the generic type is the Stack<T>. The int in Stack<int> is the type argument.

The advantage of this programming model is that the internal algorithms and data manipulation remain the same while the actual data type can change based on the way the client uses your server code.

Generics Implementation

On the surface C# generics look syntactically similar to C++ templates, but there are important differences in the way they are implemented and supported by the compiler. As you will see later in this article, this has significant implications on the manner in which you use generics.

> **Note** In the context of this article, when referring to C++ it means classic C++, not Microsoft C++ with the managed extensions.

Compared to C++ templates, C# generics can provide enhanced safety but are also somewhat limited in capabilities.

In some C++ compilers, until you use a template class with a specific type, the compiler does not even compile the template code. When you do specify a type, the compiler inserts the code inline, replacing every occurrence of the generic type parameter with the specified type. In addition, every time you use a specific type, the compiler inserts the type-specific code, regardless of whether you have already specified that type for the template class somewhere else in the application. It is up to the C++ linker to resolve this, and it is not always possible to do. This may results in code bloating, increasing both the load time and the memory footprint.

In .NET 2.0, generics have native support in IL (intermediate language) and the CLR itself. When you compile generic C# 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.

The client-side compiler uses that generic metadata to support type safety. When the client provides a specific type instead of a generic type parameter, the client's compiler substitutes the generic type parameter in the server metadata with the specified type argument. This provides the client's compiler with type-specific definition of the server, as if generics were never involved. This way the client compiler can enforce correct method parameters, type-safety checks, and even type-specific IntelliSense.

The interesting question is how does .NET compile the generic IL of the server to machine code. It turns out that the actual machine code produced depends on whether the specified types are value or reference type. If the client specifies a value type, then 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 request 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.

Generics Benefits

Generics in .NET let you reuse code and the effort you put into implementing it. The types and internal data can change without causing code bloat, regardless of whether you are using value or reference types. You can develop, test, and deploy your code once, reuse it with any type, including future types, all with full compiler support and type safety. Because the generic code does not force the boxing and unboxing of value types, or the down casting of reference types, performance is greatly improved. With value types there is typically a 200 percent performance gain, and with reference types you can expect up to a 100 percent performance gain in accessing the type (of course, the application as a whole may or may not experience any performance improvements). The source code available with this article includes a micro-benchmark application, which executes a stack in a tight loop. The application lets you experiment with value and reference types on an Object-based stack and a generic stack, as well as changing the number of loop iterations to see the effect generics have on performance.

Applying Generics

Because of the native support for generics in the IL and the CLR, most CLR-compliant language can take advantage of generic types. For example, here is some Visual Basic .NET code that uses the generic stack of **Code block 2**:

```
Dim stack As Stack(Of Integer)
stack = new Stack(Of Integer)
stack.Push(3)
Dim number As Integer
number = stack.Pop()
```

You can use generics in classes and in structs. Here is a useful generic point struct:

```
public struct Point<T>
{

   public T X;

   public T Y;
}
```

You can use the generic point for integer coordinates, for example:

```
Point<int> point;
point.X = 1;
point.Y = 2;
```

Or for charting coordinates that require floating point precision:

```
Point<double> point;
point.X = 1.2;
point.Y = 3.4;
```

Besides the basic generics syntax presented so far, C# 2.0 has some generics-specific syntax. For example, consider the **Pop()** method of **Code block 2**. Suppose instead of throwing an exception when the stack is empty, you would like to return the default value of the type stored in the stack. If you were using an Object-based stack, you would simply return null, but a generic stack could be used with value types as well. To address this issue, you can use the **default()** operator, which returns the default value of a type.

Here is how you can use default in the implementation of the **Pop()** method:

```
public T Pop()
{
   m_StackPointer--;
   if(m_StackPointer >= 0)
   {
      return m_Items[m_StackPointer];
   }
   else
   {
      m_StackPointer = 0;
      return default(T);
   }
}
```

The default value for reference types is null, and the default value for value types (such as integers, enum, and structures) is a zero whitewash (filling the structure with zeros). Consequently, if the stack is constructed with strings, the **Pop()** methods return null when the stack is empty, and when the stack is constructed with integers, the **Pop()** methods return zero when the stack is empty.

Multiple Generic Types

A single type can define multiple generic-type parameters. For example, consider the generic linked list shown in **Code block 3**.

Code block 3. Generic linked list

```
class Node<K,T>
{
   public K Key;
   public T Item;
   public Node<K,T> NextNode;
   public Node()
   {
      Key      = default(K);
      Item     = defualt(T);
      NextNode = null;
   }
   public Node(K key,T item,Node<K,T> nextNode)
   {
      Key      = key;
      Item     = item;
      NextNode = nextNode;
   }
}

public class LinkedList<K,T>
{
   Node<K,T> m_Head;
   public LinkedList()
   {
      m_Head = new Node<K,T>();
   }
   public void AddHead(K key,T item)
   {
      Node<K,T> newNode = new Node<K,T>(key,item,m_Head.NextNode);
      m_Head.NextNode = newNode;
   }
}
```

The linked list stores nodes:

```
class Node<K,T>
{...}
```

Each node contains a key (of the generic type parameter K) and a value (of the generic type parameter T). Each node also has a reference to the next node in the list. The linked list itself is defined in terms of the generic type parameters K and T:

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

This allows the list to expose generic methods like **AddHead()**:

```
public void AddHead(K key,T item);
```

Whenever you declare a variable of a type that uses generics, you must specify the types to use. However, the specified type arguments can themselves be generic type parameters. For example, the linked list has a member variable called m_Head of type Node, used for referencing the first item in the list. m_Head is declared using the list's own generic type parameters K and T

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

You need to provide type arguments when instantiating a node, and again, you can use the linked list's own generic type parameters:

```
public void AddHead(K key,T item)
{
   Node<K,T> newNode = new Node<K,T>(key,item,m_Head.NextNode);
   m_Head.NextNode = newNode;
}
```

Note that the fact the list uses the same names as the node for the generic type parameters is purely for readability purposes, and it could have used other names, such as:

```
public class LinkedList<U,V>
{...}
```

Or:

```
public class LinkedList<KeyType,DataType>
{...}
```

In which case, m_Head would have been declared as:

```
Node<KeyType,DataType> m_Head;
```

When the client is using the linked list, the client has to provide type arguments. The client can choose integers as keys and strings as data items:

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

But the client can choose any other combination, such as a time stamp for keys:

```
LinkedList<DateTime,string> list = new LinkedList<DateTime,string>();
list.AddHead(DateTime.Now,"AAA");
```

Sometimes is it useful to alias a particular combination of specific types. You can do that through the using statement, as shown in **Code block 4**. Note that the scope of aliasing is the scope of the file, so you have to repeat aliasing across the project files in the same way you would with using namespaces.

Code block 4. Generic type aliasing

```
using List = LinkedList<int,string>;

class ListClient
{
   static void Main(string[] args)
   {
      List list = new List();
      list.AddHead(123,"AAA");
   }
}
```

Generic Constraints

With C# generics, the compiler compiles the generic code into IL independent of any type arguments that the clients will use. As a result, the generic code could try to use methods, properties, or members of the generic type parameters that are incompatible with the specific type arguments the client uses. This is unacceptable because it amounts to lack of type safety. In C# you need to instruct the compiler which constraints the client-specified types must obey in order for them to be used instead of the generic type parameters. There are three types of constraints. A derivation constraint indicates to the compiler that the generic type parameter derives from a base type such an interface or a particular base class. A default constructor constraint indicates to the compiler that the generic type parameter exposes a default public constructor (a public constructor with no parameters). A reference/value type constraint constrains the generic type parameter to be a reference or a value type. A generic type can employ multiple constraints, and you even get IntelliSense reflecting the constraints when using the generic type parameter, such as suggesting methods or members from the base type.

It is important to note that although constraints are optional, they are often essential when developing a generic type. Without them, the compiler takes 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.

An example will go a long way to explain the need and use of constraints. Suppose you would like to add indexing ability or searching by key to the linked list of **Code block 3**:

```
public class LinkedList<K,T>
{
   T Find(K key)
   {...}
   public T this[K key]
   {
      get{return Find(key);}
   }
}
```

This allows the client to write the following code:

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

list.AddHead(123,"AAA");
list.AddHead(456,"BBB");
string item = list[456];
Debug.Assert(item == "BBB");
```

To implement the search, you need to scan the list, compare each node's key with the key you're looking for, and return the item of the node whose key matches. The problem is that the following implementation of **Find()** does not compile:

```
T Find(K key)
{
   Node<K,T> current = m_Head;
   while(current.NextNode != null)
   {
      if(current.Key == key) //Will not compile
         break;
      else

         current = current.NextNode;
   }
   return current.Item;
}
```

The reason is that the compiler will refuse to compile this line:

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

The line above will not compile because the compiler does not know whether K (or the actual type supplied by the client) supports the == operator. For example, structs by default do not provide such an implementation. You could try to overcome the == operator limitation by using the **IComparable** interface:

```
public interface IComparable
{
   int CompareTo(object obj);
}
```

CompareTo() returns 0 if the object you compare to is equal to the object implementing the interface, so the **Find()** method could use it as follows:

```
if(current.Key.CompareTo(key) == 0)
```

Unfortunately, this does not compile either because the compiler has no way of knowing whether K (or the actual type supplied by the client) is derived from **IComparable**.

You could explicitly cast to **IComparable** to force the compiler to compile the comparing line, except doing so is at the expense of type safety:

```
if(((IComparable)(current.Key)).CompareTo(key) == 0)
```

If the type the client uses does not derive from **IComparable**, it results in a run-time exception. In addition, when the key type used is a value type instead of the key type parameter, you force a boxing of the key, and that may have some performance implications.

Derivation Constraints

In C# 2.0 you use the **where** reserved keyword to define a constraint. Use the **where** keyword on the generic type parameter followed by a derivation colon to indicate to the compiler that the generic type parameter implements a particular interface. For example, here is the derivation constraint required to implement the **Find()** method of LinkedList:

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

You will also get IntelliSense support on the methods of interfaces you constrain.

When the client declares a variable of type **LinkedList** providing a type argument for the list's key, the client-side compiler will insist that the key type is derived from **IComparable** and will refuse to build the client code otherwise.

Note that even though the constraint allows you to use **IComparable**, it does not eliminate the boxing penalty when the key used is a value type, such as an integer. To overcome this, the System.Collections.Generic namespace defines the generic interface **IComparable<T>**:

```
public interface IComparable<T>
{
   int CompareTo(T other);
   bool Equals(T other);
}
```

You can constrain the key type parameter to support **IComparable<T>** with the key's type as the type parameter, and by doing so you gain not only type safety but also eliminate the boxing of value types when used as keys:

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

In fact, all the types that supported **IComparable** in .NET 1.1 support **IComparable<T>** in .NET 2.0. This enables the use for keys of common types such as int, string, Guid, DateTime, and so on.

In C# 2.0, all constraints must appear after the actual derivation list of the generic class. For example, if **LinkedList** derives from the **IEnumerable<T>** interface (for iterator support), you would put the **where** keyword immediately after it:

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

In general, only define a constraint at the level where you require it. In the linked list example, it is pointless to define the **IComparable<T>** derivation constraint at the node level, because the node itself does not compare keys. If you do so, you will have to place the constraint at the **LinkedList** level as well, even if the list does not compare keys. This is because the list contains a node as a member variable, causing the compiler to insist that the key type defined at the list level complies with the constraint placed by the node on the generic key type.

In other words, if you define the node as so:

```
class Node<K,T> where K : IComparable<K>
```

```
{...}
```

Then you would have to repeat the constraint at the list level, even if you do not provide the **Find()** method, or any other method for that matter:

```
public class LinkedList<KeyType,DataType> where KeyType : IComparable<KeyType>
{
   Node<KeyType,DataType> m_Head;
}
```

You can constrain multiple interfaces on the same generic type parameter, separated by a comma. For example:

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

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

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

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

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

However, at most one base class can be used in a constraint because C# does not support multiple inheritance of implementation. Obviously, the base class you constrain cannot be a sealed class or a static 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:

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

C# does allow you to specify another generic type parameter as a constraint:

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

When dealing with a derivation constraint, you can satisfy the constraint using the base type itself, not necessarily a strict sub class of it. For example:

```
public interface IMyInterface
{...}
public class MyClass<T> where T : IMyInterface
{...}
MyClass<IMyInterface> obj = new MyClass<IMyInterface>();
```

Or even:

```
public class MyOtherClass
{...}

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

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

Finally, note that when providing a derivation constraint, the base type (interface or base class) you constrain must have consistent visibility with that of the generic type parameter you define. For instance, the following constraint is valid, because internal types can use public types:

```
public class MyBaseClass
{}
internal class MySubClass<T> where T : MyBaseClass
{}
```

However, if the visibility of the two classes were reversed, such as:

```
internal class MyBaseClass
{}
public class MySubClass<T> where T : MyBaseClass
{}
```

Then the compiler would issue an error, because no client from outside the assembly will ever be able to use the generic type **MySubClass**, rendering **MySubClass** in effect as an internal rather than a public type. The reason outside clients cannot use **MySubClass** is that to declare a variable of type **MySubClass**, they require to make use of a type that derives from the internal type **MyBaseClass**.

Constructor Constraint

Suppose you want to instantiate a new generic object inside a generic class. The problem is the C# compiler does not know whether the type argument the client will use has a matching constructor, and it will refuse to compile the instantiation line.

To address this problem, C# allows you to constrain a generic type parameter such that it must support a public default constructor. This is done using the **new()** constraint. For example, here is a different way of implementing the default constructor of the generic Node <K,T> from **Code block 3**.

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

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

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

Reference/Value Type Constraint

You can constrain a generic type parameter to be a value type (such as an int, a bool, and enum) or any custom structure using the **struct** constraint:

```
public class MyClass<T> where T : struct

{...}
```

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

```
public class MyClass<T> where T : class

{...}
```

The reference/value type constraint cannot be used with base class constraint, because a base class constraint implies a class. Similarly, you cannot use the struct and the default constructor constraint, because a default constructor constraint too implies as class. Though you can use the class and the default constructor constraint, it adds no value. You can combine the reference/value type constraint with an interface constraint, as long as the reference/value type constraint appears first in the constraint list.

Generics and Casting

The C# compiler only lets you implicitly cast generic type parameters to Object, or to constraint-specified types, as shown in **Code block 5**. Such implicit casting is type safe because any incompatibility is discovered at compile-time.

Code block 5. Implicit casting of generic type parameters

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

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

```
interface ISomeInterface
{...}
class SomeClass
{...}
class MyClass<T>
{
   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:

```
class SomeClass
{...}

class MyClass<T>
{

   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 run time if the type argument used instead of the generic type parameter does not derive from the type to which you explicitly cast. Instead of risking a casting exception, a better approach is to use the `is` and `as` operators, as shown in **Code block 6**. 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. You can use `is` and `as` on both generic type parameters and on generic classes with specific type arguments.

Code block 6. Using 'is' and 'as' operators on generic type parameters

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

Inheritance and Generics

When deriving from a generic base class, you must provide a type argument instead of the base-class's generic type parameter:

```
public class BaseClass<T>
{...}
public class SubClass : BaseClass<int>
{...}
```

If the subclass is generic, instead of a concrete type argument, you can use the subclass generic type parameter as the specified type for the generic base class:

```
public class SubClass<T> : BaseClass<T>
{...}
```

When using the subclass generic type parameters, you must repeat any constraints stipulated at the base class level at the subclass level. For example, derivation constraint:

```
public class BaseClass<T>  where T : ISomeInterface
{...}
public class SubClass<T> : BaseClass<T> where T : ISomeInterface
{...}
```

Or constructor constraint:

```
public class BaseClass<T>  where T : new()
{
   public T SomeMethod()
   {
      return new T();
   }
}
public class SubClass<T> : BaseClass<T> where T : new()
{...}
```

A base class can define virtual methods whose signatures use generic type parameters. When overriding them, the subclass must provide the corresponding types in the method signatures:

```
public class BaseClass<T>
{
   public virtual T SomeMethod()
   {...}
}
public class SubClass: BaseClass<int>
{
   public override int SomeMethod()
   {...}
}
```

If the subclass is generic it can also use its own generic type parameters for the override:

```
public class SubClass<T>: BaseClass<T>
{
   public override T SomeMethod()
   {...}
}
```

You can define generic interfaces, generic abstract classes, and even generic abstract methods. These types behave like any other generic base type:

```
public interface ISomeInterface<T>
```

```
{
   T SomeMethod(T t);
}
public abstract class BaseClass<T>
{
   public abstract T SomeMethod(T t);
}

public class SubClass<T> : BaseClass<T>
{
   public override T SomeMethod(T t)
   {...)
}
```

There is an interesting use for generic abstract methods and generic interfaces. In C# 2.0, it is impossible to use operators such as + or += on generic type parameters. For example, the following code does not compile because C# 2.0 does not have operator constraints:

```
public class Calculator<T>
{
   public T Add(T arg1,T arg2)
   {
      return arg1 + arg2;//Does not compile
   }
   //Rest of the methods
}
```

Nonetheless, you can compensate using abstract methods (or preferably interfaces) by defining generic operations. Since an abstract method cannot have any code in it, you can specify the generic operations at the base class level, and provide a concrete type and implementation at the subclass level:

```
public abstract class BaseCalculator<T>
{
   public abstract T Add(T arg1,T arg2);
   public abstract T Subtract(T arg1,T arg2);
   public abstract T Divide(T arg1,T arg2);
   public abstract T Multiply(T arg1,T arg2);
}
public class MyCalculator : BaseCalculator<int>
{
   public override int Add(int arg1, int arg2)
   {
      return arg1 + arg2;
   }
   //Rest of the methods
}
```

A generic interface will yield a somewhat cleaner solution as well:

```
public interface ICalculator<T>
{
   T Add(T arg1,T arg2);
   //Rest of the methods
}
public class MyCalculator : ICalculator<int>
{
   public int Add(int arg1, int arg2)
   {
      return arg1 + arg2;
   }
   //Rest of the methods
}
```

Generic Methods

In C# 2.0, a method can define generic type parameters, specific to its execution scope:

```
public class MyClass<T>
{
   public void MyMethod<X>(X x)
   {...}
}
```

This is an important capability because it allows you to call the method with a different type every time, which is very handy for utility classes.

You can define method-specific generic type parameters even if the containing class does not use generics at all:

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

This ability is for methods only. Properties or indexers can only use generic type parameters defined at the scope of the class.

When calling a method that defines generic type parameters, you can provide the type to use at the call site:

```
MyClass obj = new MyClass();
obj.MyMethod<int>(3);
```

That said, when the method is invoked the C# compiler is smart enough to infer the correct type based on the type of parameter passed in, and it allows omitting the type specification altogether:

```
MyClass obj = new MyClass();
obj.MyMethod(3);
```

This ability is called generic type inference. Note that the compiler cannot infer the type based on the type of the returned value alone:

```
public class MyClass
{
   public T MyMethod<T>()
   {}
}
MyClass obj = new MyClass();
int number = obj.MyMethod();//Does not compile
```

When a method defines its own generic type parameters, it can also define constraints for these types:

```
public class MyClass
{
   public void SomeMethod<T>(T t) where T : IComparable<T>
   {...}
}
```

However, you cannot provide method-level constraints for class-level generic type parameters. All constraints for class-level generic type parameters must be defined at the class scope.

When overriding a virtual method that defines generic type parameters, the subclass method must redefine the method-specific generic type parameter:

```
public class BaseClass
{
   public virtual void SomeMethod<T>(T t)
   {...}
}
public class SubClass : BaseClass
{
   public override void SomeMethod<T>(T t)
   {...}
}
```

The subclass implementation must repeat all constraints that appeared at the base method level:

```
public class BaseClass
{
   public virtual void SomeMethod<T>(T t) where T : new()
   {...}
}
public class SubClass : BaseClass
{
   public override void SomeMethod<T>(T t) where T : new()
   {...}
}
```

Note that the method override cannot define new constraints that did not appear at the base method.

In addition, if the subclass method calls the base class implementation of the virtual method, it must specify the type argument to use instead of the generic base method type parameter. You can either explicitly specify it yourself or rely on type inference if it is available:

```
public class BaseClass
{
   public virtual void SomeMethod<T>(T t)
   {...}
}
public class SubClass : BaseClass
{
   public override void SomeMethod<T>(T t)
   {
      base.SomeMethod<T>(t);
      base.SomeMethod(t);
   }
}
```

Generic Static Method

C# allows you to define static methods that use generic type parameters. However, when invoking such a static method, you need to provide the concrete type for the containing class at the call site, such as in this example:

```
public class MyClass<T>
{

   public static T SomeMethod(T t)

   {...}
}
int number = MyClass<int>.SomeMethod(3);
```

Static methods can define method-specific generic type parameters and constraints, similar to instance methods. When calling such methods, you need to provide the method-specific types at the call site, either explicitly:

```
public class MyClass<T>
{
   public static T SomeMethod<X>(T t,X x)
   {..}
}
int number = MyClass<int>.SomeMethod<string>(3,"AAA");
```

Or rely on type inference when possible:

```
int number = MyClass<int>.SomeMethod(3,"AAA");
```

Generic static methods are subjected to all constraints imposed on the generic type parameter they use at the class level. As with instance method, you can provide constraints for generic type parameters defined by the static method:

```
public class MyClass
{
   public static T SomeMethod<T>(T t) where T : IComparable<T>
   {...}
}
```

Operators in C# are nothing more than static methods and C# allows you to overload operators for your generic types. Imagine that the generic **LinkedList** of **Code block 3** provided the + operator for concatenating linked lists. The + operator enables you to write the following elegant code:

```
LinkedList<int,string> list1 = new LinkedList<int,string>();
LinkedList<int,string> list2 = new LinkedList<int,string>();
 ...
LinkedList<int,string> list3 = list1+list2;
```

Code block 7 shows the implementation of the generic + operator on the **LinkedList** class. Note that operators cannot define new generic type parameters.

Code block 7. Implementing a generic operator

```
public class LinkedList<K,T>
{
   public static LinkedList<K,T> operator+(LinkedList<K,T> lhs,
                                        LinkedList<K,T> rhs)
   {
      return concatenate(lhs,rhs);
   }
   static LinkedList<K,T> concatenate(LinkedList<K,T> list1,
                                   LinkedList<K,T> list2)
   {
      LinkedList<K,T> newList = new LinkedList<K,T>();
      Node<K,T> current;
      current = list1.m_Head;
      while(current != null)
      {
         newList.AddHead(current.Key,current.Item);
         current = current.NextNode;
      }
      current = list2.m_Head;
      while(current != null)
      {
         newList.AddHead(current.Key,current.Item);
         current = current.NextNode;
      }
      return newList;
   }
   //Rest of LinkedList
}
```

Generic Delegates

A delegate defined in a class can take advantage of the generic type parameter of that class. For example:

```
public class MyClass<T>
{
   public delegate void GenericDelegate(T t);
   public void SomeMethod(T t)
   {...}
}
```

When specifying a type for the containing class, it affects the delegate as well:

```
MyClass<int> obj = new MyClass<int>();
MyClass<int>.GenericDelegate del;

del = new MyClass<int>.GenericDelegate(obj.SomeMethod);
del(3);
```

C# 2.0 allows you to make a direct assignment of a method reference into a delegate variable:

```
MyClass<int> obj = new MyClass<int>();
MyClass<int>.GenericDelegate del;

del = obj.SomeMethod;
```

I am calling this feature *delegate inference*. The compiler is capable of inferring the type of the delegate you assign into, finding if the target object has a method by the name you specify, and verifying that the method's signature matches. Then, the compiler creates a new delegate of the inferred argument type (including the correct type instead of the generic type parameter), and assigns the new delegate into the inferred delegate.

Like classes, structs, and methods, delegates can define generic type parameters too:

```
public class MyClass<T>
{
   public delegate void GenericDelegate<X>(T t,X x);
}
```

Delegates defined outside the scope of a class can use generic type parameters. In that case, you have to provide the type arguments for the delegate when declaring and instantiating it:

```
public delegate void GenericDelegate<T>(T t);

public class MyClass
{
   public void SomeMethod(int number)
   {...}
}

MyClass obj = new MyClass();
GenericDelegate<int> del;

del = new GenericDelegate<int>(obj.SomeMethod);
del(3);
```

Alternatively, you can use delegate inference when assigning the delegate:

```
MyClass obj = new MyClass();
GenericDelegate<int> del;

del = obj.SomeMethod;
```

And naturally, a delegate can define constraints to accompany its generic type parameters:

```
public delegate void MyDelegate<T>(T t) where T : IComparable<T>;
```

The delegate-level constraints are enforced only on the using side, when declaring a delegate variable and instantiating a delegate object, similar to any other constraint at the scope of types or methods.

Generic delegates are especially useful when it comes to events. You can literally define a limited set of generic delegates, distinguished only by the number of the generic type parameters they require, and use these delegates for all of your event handling needs. **Code block 8** demonstrates the use of a generic delegate and a generic event-handling method.

Code block 8. Generic event handling

```
public delegate void GenericEventHandler<S,A>(S sender,A args);
public class MyPublisher
{
   public event GenericEventHandler<MyPublisher,EventArgs> MyEvent;
   public void FireEvent()
   {
      MyEvent(this,EventArgs.Empty);
   }
}
public class MySubscriber<A> //Optional: can be a specific type
{
   public void SomeMethod(MyPublisher sender,A args)
   {...}
}
MyPublisher publisher = new MyPublisher();
MySubscriber<EventArgs> subscriber = new MySubscriber<EventArgs>();
publisher.MyEvent += subscriber.SomeMethod;
```

Code block 8 uses a generic delegate called **GenericEventHandler**, which accepts a generic sender type and a generic type parameter. Obviously, if you need more parameters, you can simply add more generic type parameters, but I wanted to model **GenericEventHandler** after the non-generic .NET **EventHandler**, defined as:

```
public void delegate EventHandler(object sender,EventArgs args);
```

Unlike **EventHandler**, **GenericEventHandler** is type safe, as shown in **Code block 8** because it accepts only objects of the type **MyPublisher** as senders, rather than mere Object. In fact, .NET already defines a generic flavor of **EventHandler** in the System namespace:

```
public void delegate EventHandler(object sender,A args) where A : EventArgs;
```

Generics and Reflection

In .NET 2.0, reflection is extended to support generic type parameters. The type Type can now represent generic types with specific type arguments (called *bounded types*), or unspecified (*unbounded*) types. As in C# 1.1, you can obtain the Type of any type by using the **typeof** operator or by calling the **GetType()** method that every type supports. Regardless of the way you choose, both yield the same Type. For example, in the following code sample, type1 is identical to **type2**.

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

Type type1 = typeof(LinkedList<int,string>);
Type type2 = list.GetType();
Debug.Assert(type1 == type2);
```

Both **typeof** and **GetType()** can operate on generic 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 unbounded generic types. 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 unbounded 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]
```

Type has new methods and properties designed to provide reflection information about the generic aspect of the type. **Code block 9** shows the new methods.

Code block 9. Type's generic reflection members

```
public abstract class Type : //Base types
{
   public virtual bool ContainsGenericParameters{get;}
   public virtual int GenericParameterPosition{get;}
   public virtual bool HasGenericArguments{get;}
   public virtual bool IsGenericParameter{get;}
   public virtual bool IsGenericTypeDefinition{get;}
   public virtual Type BindGenericParameters(Type[] typeArgs);
   public virtual Type[] GetGenericArguments();
   public virtual Type GetGenericTypeDefinition();
   //Rest of the members
}
```

The most useful of these new members are the **HasGenericArguments** property and the **GetGenericArguments ()** and **GetGenericTypeDefinition()** methods. The rest of Type's new members are for advanced and somewhat esoteric scenarios beyond the scope of this article.

As its name indicates, **HasGenericArguments** 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. **Code block 10** demonstrates using these generic-handling Type members to obtain generic reflection information on the **LinkedList** from **Code block 3**.

Code block 10. Using Type for generic reflection

```
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.HasGenericArguments);

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();
Debug.Assert(unboundedType == typeof(LinkedList<,>));
Trace.WriteLine(unboundedType.ToString());
//Writes: LinkedList`2[K,T]
```

Similar to Type, **MethodInfo** and its base class **MethodBase** have new members that reflect generic method information.

As in C# 1.1, you can use **MethodInfo** (as well as a number of other options) for late binding invocation. However, the type of the parameters you pass for the late binding must match the bounded types used instead of the generic type parameters (if any):

```
LinkedList<int,string> list = new LinkedList<int,string>();
Type type = list.GetType();
MethodInfo methodInfo = type.GetMethod("AddHead");
object[] args = {1,"AAA"};
methodInfo.Invoke(list,args);
```

Attributes and Generics

When defining an attribute, you can instruct the compiler that the attribute should target generic type parameters, using the new **GenericParameter** value of the enum **AttributeTargets**:

```
[AttributeUsage(AttributeTargets.GenericParameter)]
public class SomeAttribute : Attribute
```

```
{...}
```

Note that C# 2.0 does not allow you to define generic attributes.

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

Yet internally, an attribute class can take advantage of generics by using generic types or define helper generic methods, like any other type:

```
public class SomeAttribute : Attribute
{
   void SomeMethod<T>(T t)
   {...}
   LinkedList<int,string> m_List = new LinkedList<int,string>();
}
```

Generics and the .NET Framework

To conclude this article, here are how some other areas in .NET besides C# itself are taking advantage of or interacting with generics.

System.Array and Generics

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. **Code Block 11** is a partial listing of these static methods.

Code block 11. The generic methods of System.Array

```
public abstract class Array
{
   //Partial listing of the static methods:
   public static IList<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<K,V>(K[] keys, V[] items,
                             IComparer<K> comparer);
   public static void Sort<T>(T[] array,Comparison<T> comparison)
}
```

System.Array's static generic methods all work with the following four generic delegates defined in the **System** namespace:

```
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);
```

Code block 12 demonstrates using these generic methods and delegates. It initializes an array with all the integers from 1 to 20. Then, using an anonymous method and the **Action<T>** delegate, the code traces these numbers using the **Array.ForEach()** method. Using a second anonymous method and the **Predicate<T>** delegate, the code finds all the prime numbers in the array by calling the **Array.FindAll()** method, which returns another array of the same generic type. Finally, the prime numbers are traced using the same **Action<T>** delegate and anonymous method. Note the use of type parameter inference in **Code block 12.** You can use the static methods without specifying the type parameters.

Code block 12. Using the generic methods of System.Array

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
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;
                              }
                           };
Array.ForEach(numbers,trace);
int[] primes = Array.FindAll(numbers,isPrime);
Array.ForEach(primes,trace);
```

Similar generic methods are also available on the class **List<T>** defined in the **System.Collections.Generic** namespace. These methods use the same four generic delegates. In fact, you can take advantage of those delegates in your code too, as shown in the following section.

The Static Collection Class

While both **System.Array** and **List<T>** offer handy utility methods that greatly simplify working with them, .NET does not offer such support for other collections. To compensate, the source code available with this article includes the static helper class **Collection**, defined as:

```
public static class Collection
{
   public static IList<T> AsReadOnly<T>(IEnumerable<T> collection);
   public static U[] ConvertAll<T,U>(IEnumerable<T> collection,
                                   Converter<T,U> converter);
   public static bool Contains<T>(IEnumerable<T> collection,T item)
                                             where T : IComparable<T>;
   public static bool Exists<T>(IEnumerable<T> collection,Predicate<T>);
   public static T Find<T>(IEnumerable<T> collection,Predicate<T> match);
   public static T[] FindAll<T>(IEnumerable<T> collection,
                              Predicate<T> match);
   public static int FindIndex<T>(IEnumerable<T> collection,T value)
                                              where T : IComparable<T>;
   public static T FindLast<T>(IEnumerable<T> collection,
                             Predicate<T> match);
   public static int FindLastIndex<T>(IEnumerable<T> collection,T value)
                                              where T : IComparable<T>;
   public static void ForEach<T>(IEnumerable<T> collection,Action<T> action);
   public static T[] Reverse<T>(IEnumerable<T> collection);
   public static T[] Sort<T>(IEnumerable<T> collection);
   public static T[] ToArray<T>(IEnumerable<T> collection);
   public static bool TrueForAll<T>(IEnumerable<T> collection,
                                  Predicate<T> match);
   //Overloaded versions for IEnumerator<T>
}
```

The implementation of **Collection** is straightforward. For example, here is the **ForEach<T>()** method:

```
public static void ForEach<T>(IEnumerator<T> iterator,Action<T> action)
{
   /* Some parameter checking here, then: */
   while(iterator.MoveNext())
   {
      action(iterator.Current);
   }
}
```

The **Collection** static class is used very similar to both **Array** and **List<T>**, utilizing the same generic delegates. You can use **Collection** with any collection, as long as that collection supports either **IEnumerable** or **IEnumerator**:

```
Queue<int> queue = new Queue<int>();
//Some code to initialize queue
Action<int> trace =  delegate(int number)
                     {
                        Trace.WriteLine(number);
                     };
Collection.ForEach(queue,trace);
```

Generic Collections

The data structures in System.Collections are all Object-based, and thus inheriting the two problems described at the beginning of this article, namely, inferior performance and lack of type safety. .NET 2.0 introduces a set of generic collections in the System.Collections.Generic namespace. For example, there are generic **Stack<T>** and a generic **Queue<T>** classes. The **Dictionary<K,T>** data structure is equivalent to the non-generic **HashTable**, and there is also a **SortedDictionary<K,T>** class somewhat like **SortedList**. The class **List<T>** is analogous to the non-generic **ArrayList**. Table 1 maps the main types of System.Collections.Generic to those of System.Collections.

Table 1. Mapping System.Collections.Generic to System.Collections

System.Collections.Generic	System.Collections
Comparer<T>	Comparer
Dictionary<K,T>	HashTable
LinkedList<T>	-
List<T>	ArrayList
Queue<T>	Queue
SortedDictionary<K,T>	SortedList
Stack<T>	Stack
ICollection<T>	ICollection
IComparable<T>	System.IComparable

IDictionary<K,T>	IDictionary
IEnumerable<T>	IEnumerable
IEnumerator<T>	IEnumerator
IList<T>	IList

All the generic collections in System.Collections.Generic also implement the generic **IEnumerable<T>** interface, defined as:

```
public interface IEnumerable<T>
{
   IEnumerator<T> GetEnumerator();
}
public interface IEnumerator<T> : IDisposable
{
   T Current{get;}
   bool MoveNext();
}
```

Briefly, **IEnumerable<T>** provides access to the **IEnumerator<T>** iterator interface, used for abstracted iterations over the collection. All the collections implement **IEnumerable<T>** on a nested struct, where the generic type parameter T is the type the collection stores.

Of particular interest is the way the dictionary collections define their iterators. A dictionary is actually a collection of not one but two types of generic parameter—the key and the value. System.Collection.Generic provides a generic struct called KeyValuePair<K,V> defined as:

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

KeyValuePair<K,V> simply stores a pair of a generic key and a generic value. This struct is what the dictionary manages as a collection, and what it uses for its implementation of **IEnumerable<T>**. The **Dictionary** class specifies the generic KeyValuePair<K,V> structure as the item argument for **IEnumerable<T>** and **ICollection<T>**:

```
public class Dictionary<K,T> : IEnumerable<KeyValuePair<K,T>>,
                               ICollection<KeyValuePair<K,T>>,
                               //More interfaces
{...}
```

The key and value type parameters used in KeyValuePair<K,V> are of course the dictionary's own generic key and value type parameters. You can certainly do the same in your own generic data structures that use pairs of keys and values. For example:

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

Serialization and Generics

.NET allows you to have serializable generic types:

```
[Serializable]
public class MyClass<T>
{...}
```

When serializing a type, besides the state of the object members, .NET persists metadata about the object and its type. If the serializable type is generic and it contains bounded types, the metadata about the generic type contains type information about the bounded types as well. Consequently, each permutation of a generic type with a specific argument type is considered a unique type. For example, you cannot serialize an object type **MyClass<int>** but deserialize it into an object of type **MyClass<string>**. Serializing an instance of a generic type is no different from serializing a non-generic type. However, when deserializing that type, you need to declare a variable with matching specific types, and specify these types again when down casting the Object returned from Deserialize. **Code block 13** shows serialization and deserialization of a generic type.

Code block 13. Client-side serialization of a generic type

```
[Serializable]
public class MyClass<T>
{...}
MyClass<int> obj1 = new MyClass<int>();

IFormatter formatter = new BinaryFormatter();
Stream stream = new FileStream("obj.bin",FileMode.Create,FileAccess.ReadWrite);
using(stream)
{
   formatter.Serialize(stream,obj1);
   stream.Seek(0,SeekOrigin.Begin);

   MyClass<int> obj2;
   obj2 = (MyClass<int>)formatter.Deserialize(stream);
}
```

Note that **IFormatter** is object-based. You can compensate by defining a generic version of **IFormatter**:

```
public interface IGenericFormatter
{
   T Deserialize<T>(Stream serializationStream);
   void Serialize<T>(Stream serializationStream,T graph);
}
```

You can implement **IGenericFormatter** by containing an-object based formatter:

```
public class GenericFormatter<F> : IGenericFormatter
                                   where F : IFormatter,new()
{
   IFormatter m_Formatter = new F();
   public T Deserialize<T>(Stream serializationStream)
   {
      return (T)m_Formatter.Deserialize(serializationStream);
   }
   public void Serialize<T>(Stream serializationStream,T graph)
   {
      m_Formatter.Serialize(serializationStream,graph);
   }
}
```

Note the use of the two constraints on the generic type parameter F. While it is possible to use **GenericFormatter<F>** as-is:

```
using GenericBinaryFormatter = GenericFormatter<BinaryFormatter>;
using GenericSoapFormatter2  = GenericFormatter<SoapFormatter>;
```

You can also strongly-type the formatter to use:

```
public sealed class GenericBinaryFormatter : GenericFormatter<BinaryFormatter>
{}
public sealed class GenericSoapFormatter : GenericFormatter<SoapFormatter>
{}
```

The advantage of a strongly-typed definition is that you can share it across files and assemblies, as opposed to the **using** alias.

Code block 14 is the same as **Code block 13** expect it uses the generic formatter:

Code block 14. Using IGenericFormatter

```
[Serializable]
public class MyClass<T>
{...}
MyClass<int> obj1 = new MyClass<int>();

IGenericFormatter formatter = new GenericBinaryFormatter();
Stream stream = new FileStream("obj.bin",FileMode.Create,FileAccess.ReadWrite);
using(stream)
{
   formatter.Serialize(stream,obj1);
   stream.Seek(0,SeekOrigin.Begin);
   MyClass<int> obj2;
   obj2 = formatter.Deserialize(stream);
}
```

Generics and Remoting

You can define and deploy remote classes that utilize generics, and you can use programmatic or administrative configuration. Consider the class **MyServer<T>** that uses generics and is derived from **MarshalByRefObject**:

```
public class MyServer<T> : MarshalByRefObject
{...}
```

You can only access this class over remoting when the type parameter **T** is a marshalable object. This implies that **T** is either a serializable type or is derived from **MarshalByRefObject**. You can enforce that by constraining **T** to derive from **MarshalByRefObject**:

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

When using administrative type registration, you need to specify the exact type arguments to use instead of the generic type parameters. You must name the types in a language-neutral manner, and provide fully qualified namespaces. For example, suppose the class **MyServer<T>** is defined in the namespace **RemoteServer** in the assembly **ServerAssembly**, and you want to use it with integer instead of the generic type parameter T, in client activated mode. In that case, the required client-side type registration entry in the configuration file would be:

```
<client url="...some url goes here...">
   <activated type="RemoteServer.MyServer[[System.Int32]],ServerAssembly"/>
</client>
```

The matching host-side type registration entry in the configuration file is:

```
<service>
   <activated type="RemoteServer.MyServer[[System.Int32]],ServerAssembly"/>
</service>
```

The double square brackets are used to specify multiple types. For example:

```
LinkedList[[System.Int32],[System.String]]
```

When using programmatic configuration, you configure activation modes and type registration similar to C# 1.1, except when defining the type of the remote object you have to provide type arguments instead of the generic type parameters. For example, for host-side activation mode and type registration you would write:

```
Type serverType = typeof(MyServer<int>);
RemotingConfiguration.RegisterActivatedServiceType(serverType);
```

For client-side type activation mode and location registration, you have the following:

```
Type serverType = typeof(MyServer<int>);
string url = ...; //some url initialization
RemotingConfiguration.RegisterWellKnownClientType(serverType,url);
```

When instantiating the remote server, simply provide the type arguments, just as if you where using a local generic type:

```
MyServer<int> obj;
obj = new MyServer<int>();
//Use obj
```

Instead of using new, the client can optionally use the methods of the **Activator** class to connect to remote objects. When using **Activator.GetObject()** you need to provide the type arguments to use, and provide the argument types when explicitly casting the returned Object:

```
string url = ...; //some url initialization
Type serverType = typeof(MyServer<int>);
MyServer<int> obj;
obj = (MyServer<int>)Activator.GetObject(serverType,url);
//Use obj
```

You can also use **Activator.CreateInstance()** with generic types:

```
Type serverType = typeof(MyServer<int>);
MyServer<int> obj;
obj = (MyServer<int>)Activator.CreateInstance(serverType);
//Use obj
```

In fact, **Activator** also offers a generic version of **CreateInstance()**, defined as:

```
T CreateInstance<T>();
```

CreateInstance<T>() is used similar to the non-generic flavor, only with the added benefit of type safety:

```
Type serverType = typeof(MyServer<int>);
MyServer<int> obj;
obj = Activator.CreateInstance(serverType);
//Use obj
```

What Generics Cannot Do

Under .NET 2.0, you cannot define generic Web services. That is, Web methods that use generic type parameters. The reason is that none of the Web service standards support generic services.

You also cannot use generic types on a serviced component. The reason is that generics do not meet COM visibility requirements, which are required for serviced components (just like you could not use C++ templates in COM or COM+).

Conclusion

C# generics are an invaluable part of your development arsenal. They improve performance, type safety and quality, reduce repetitive programming tasks, simplify the overall programming model, and do so with elegant, readable syntax. While the roots of C# generics are C++ templates, C# takes generics to a new level by providing compile-time safety and support. C# utilizes two-phase compilation, metadata, and innovative concepts such as constraints and generic methods. No doubt future versions of C# will continue to evolve generics, adding new capabilities and extending generics to other areas of .NET Framework such as data access or localization.

Juval Lowy is a software architect and the principal of IDesign, a consulting and training company. 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). The book is dedicated to component-oriented programming and design, as well as the related system issues. Juval participates in the Microsoft internal design reviews for future versions of .NET and is a frequent presenter at development conferences. Microsoft recognized Juval as a Software Legend and one of the world's top .NET experts and industry leaders. Contact Juval at http://www.idesign.net.

Did you find this helpful? ◯ Yes ◯ No