Constraints on Type Parameters

When you define a generic class, you can apply restrictions to the kinds of types that client code can use for type arguments when it instantiates your class. If client code tries to instantiate your class by using a type that is not allowed by a constraint, the result is a compile-time error. These restrictions are called constraints. Constraints are specified by using the where contextual keyword. The following table lists the six types of constraints:

Constraint	Description
where T: struct	The type argument must be a value type. Any value type except Nullable can be specified. See Using Nullable Types (C# Programming Guide) for more information.
where T : class	The type argument must be a reference type; this applies also to any class, interface, delegate, or array type.
where T : new()	The type argument must have a public parameterless constructor. When used together with other constraints, the new() constraint must be specified last.
where T : <base class name>	The type argument must be or derive from the specified base class.
where T : <interface name>	The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic.
where T : U	The type argument supplied for T must be or derive from the argument supplied for U.

Why Use Constraints?

If you want to examine an item in a generic list to determine whether it is valid or to compare it to some other item, the compiler must have some guarantee that the operator or method it has to call will be supported by any type argument that might be specified by client code. This guarantee is obtained by applying one or more constraints to your generic class definition.

The following code example demonstrates the functionality we can add to the GenericList<T> class.

```
public class Employee
{
    private string name;
    private int id;

    public Employee(string s, int i)
    {
        name = s;
        id = i;
    }

    public string Name
    {
```

```
        get { return name; }
        set { name = value; }
    }

    public int ID
    {
        get { return id; }
        set { id = value; }
    }
}
public class GenericList<T> where T : Employee
{
    private class Node
    {
        private Node next;
        private T data;

        public Node(T t)
        {
            next = null;
            data = t;
        }

..............

public T FindFirstOccurrence(string s)
    {
        Node current = head;
        T t = null;

        while (current != null)
        {
            //The constraint enables access to the Name property.
            if (current.Data.Name == s)
            {
                t = current.Data;
                break;
            }
            else
            {
                current = current.Next;
            }
        }
        return t;
    }
}
```

The constraint enables the generic class to use the Employee.Name property because all items of type T are guaranteed to be an Employee object.

new Constraint

The new constraint specifies that any type argument in a generic class declaration must have a public parameterless constructor. To use the new constraint, the type cannot be abstract.

Apply the new constraint to a type parameter when your generic class creates new instances of the type, as shown in the following example:

```
class ItemFactory<T> where T : new()
    {
        public T GetNewItem()
        {
            return new T();
        }
    }
```

When you use the new() constraint with other constraints, it must be specified last:

```
public class ItemFactory2<T>
        where T : IComparable, new()
    {
    }
```

Generic Interfaces

When an interface is specified as a constraint on a type parameter, only types that implement the interface can be used. The following code example shows a SortedList<T> class that derives from the GenericList<T> class.

```
namespace Example
{
  class Program
  {
    class Data<T>
      where T : IDisposable      // Generic Constraint interface
    {
      public void Test()
      {
        Console.WriteLine("Data Table");  // Data Table
      }
    }

    static void Main()
    {
      Data<DataTable> d = new Data<DataTable>();
      d.Test();
    }
```

```
    }
}
```

In the above code, we are using a generic interface constraint. DataTable (derived from System.Data) is implementing the IDisposable (interface).

```
Another example:

using System;
using System.Collections.Generic;

public class GenericComparer<T> : IComparer<T>
    where T : IComparable<T>
{
    public int Compare(T x, T y)
    {
        return x.CompareTo(y);
    }
}
```

The example below shows a generic class declaration where the parameterized type has to implement a specific interface.

```
interface ICustomInterface
{ }
public class Class2 : ICustomInterface

public class CustomList<T> where T : ICustomInterface
{
    void AddToList(T element)
    {}
}

public class Class1
{
    public Class1()
    {
        CustomList<string> myCustomList1 = new CustomList<string>();
        CustomList<Class2> myCustomList1 = new CustomList<Class2>();
    }
}
```

We obtained an error with types that don't implement "ICustomInterface" like string.

where T : class

Class is a reference type. So it is a reference type constraint. it can be applied to any class, interface, delegate and array type.

```
namespace Example
{
  class Program
  {
```

```
    class MyClass
    {
      public void Test<T>()
        where T : class     // Generic Constraint
      {
        Console.WriteLine("Hello"); // Prints Hello
      }
    }

    static void Main()
    {
      MyClass my = new MyClass();
      my.Test<string>();
    }
  }
}
```

Constraining Multiple Parameters

You can apply constraints to multiple parameters, and multiple constraints to a single parameter, as shown in the following example:

```
class Base { }
class Test<T, U>
    where U : struct
    where T : Base, new() { }
```

Type Parameters as Constraints

The use of a generic type parameter as a constraint is useful when a member function with its own type parameter has to constrain that parameter to the type parameter of the containing type, as shown in the following example:

```
class List<T>
{
    void Add<U>(List<U> items) where U : T {/*...*/}
}
```

In the previous example, T is a type constraint in the context of the **Add** method, and an unbounded type parameter in the context of the **List** class.