Constraints on generic type parameters (T)

Constraints inform the compiler about the capabilities a type argument must have. Without any constraints, the type argument could be any type. The compiler can only assume the members of System.Object, which is the ultimate base class for any .NET type. If client code uses a type that doesn't satisfy a constraint, the compiler issues an error. Constraints are specified by using the **`where`** contextual keyword. The following table lists the various types of constraints:

1. **`where T : struct`**
   The type argument must be a non-nullable value type. Because all value types have an accessible parameterless constructor, the struct constraint implies the new() constraint and can't be combined with the new() constraint. You can't combine the struct constraint with the unmanaged constraint.
2. **`where T : class`**
   The type argument must be a reference type. This constraint applies also to any class, interface, delegate, or array type. In a nullable context in C# 8.0 or later, T must be a non-nullable reference type.
3. **`where T : class?`**
   The type argument must be a reference type, either nullable or non-nullable. This constraint applies also to any class, interface, delegate, or array type.
4. **`where T : notnull`**
   The type argument must be a non-nullable type. The argument can be a non-nullable reference type in C# 8.0 or later, or a non-nullable value type.
5. **`where T : unmanaged`**
   The type argument must be a non-nullable unmanaged type. The unmanaged constraint implies the struct constraint and can't be combined with either the struct or new() constraints.
6. **`where T : new()`**
   The type argument must have a public parameter less constructor. When used together with other constraints, the new() constraint must be specified last. The new() constraint can't be combined with the struct and unmanaged constraints.
7. **`where T : <base class name>`**
   The type argument must be or derive from the specified base class. In a nullable context in C# 8.0 and later, T must be a non-nullable reference type derived from the specified base class.
8. **`where T : <base class name>?`**
   The type argument must be or derive from the specified base class. In a nullable context in C# 8.0 and later, T may be either a nullable or non-nullable type derived from the specified base class.
9. **`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. In a nullable context in C# 8.0 and later, T must be a non-nullable type that implements the specified interface.
10. **`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. In a nullable context in C# 8.0, T

may be a nullable reference type, a non-nullable reference type, or a value type. T may not be a nullable value type.

11. **`where T : U`**

    The type argument supplied for T must be or derive from the argument supplied for U. In a nullable context, if U is a non-nullable reference type, T must be non-nullable reference type. If U is a nullable reference type, T may be either nullable or non-nullable.

Example:

```
public class Employee
{
    public Employee(string name, int id) => (Name, ID) = (name, id);
    public string Name { get; set; }
    public int ID { get; set; }
}
public class GenericList<T> where T : Employee
{
//Consider code for this class given in example of previous article
    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;
    }
}
```

Constraining multiple parameters

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

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

Unbounded type parameters

Type parameters that have no constraints, such as T in `public class SampleClass<T>{}`, are called unbounded type parameters. Unbounded type parameters have the following rules:

- The != and == operators can't be used because there's no guarantee that the concrete type argument will support these operators.
- They can be converted to and from `System.Object` or explicitly converted to any `interface` type.
- You can compare them to `null`. If an unbounded parameter is compared to null, the comparison will always return false if the type argument is a value type.

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:

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

Unmanaged constraint

Beginning with C# 7.3, you can use the unmanaged constraint to specify that the type parameter must be a non-nullable unmanaged type. The unmanaged constraint enables you to write reusable routines to work with types that can be manipulated as blocks of memory, as shown in the following:

```
unsafe public static byte[] ToByteArray<T>(this T argument) where T : unmanaged
{
    var size = sizeof(T);
    var result = new Byte[size];
    Byte* p = (byte*)&argument;
    for (var i = 0; i < size; i++)
        result[i] = *p++;
    return result;
}
```

Enum constraints

Beginning in C# 7.3, you can also specify the `System.Enum` type as a base class constraint. The CLR always allowed this constraint, but the C# language disallowed it. Generics using `System.Enum` provide type-safe programming to cache results from using the static methods in `System.Enum`. The following sample finds all the valid values for an enum type, and then builds a dictionary that maps those values to its string representation.

```
public static Dictionary<int, string> EnumNamedValues<T>() where T : System.Enum
{
```

```
    var result = new Dictionary<int, string>();
    var values = Enum.GetValues(typeof(T));
    foreach (int item in values)
        result.Add(item, Enum.GetName(typeof(T), item));
    return result;
}
```

Delegate constraints

Also beginning with C# 7.3, you can use System.Delegate or System.MulticastDelegate as a base class constraint. The CLR always allowed this constraint, but the C# language disallowed it. The System.Delegate constraint enables you to write code that works with delegates in a type-safe manner. The following code defines an extension method that combines two delegates provided they're the same type:

```
public static TDelegate TypeSafeCombine<TDelegate>(this TDelegate source,
TDelegate target)
    where TDelegate : System.Delegate
    => Delegate.Combine(source, target) as TDelegate;
```