Constraints on type parameters (C# Programming Guide)

04/12/2018 • 8 minutes to read • Contributors all

In this article

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 Object, which is the ultimate base class for any .NET type. For more information, see Why use constraints. 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. Constraints are specified by using the `where` contextual keyword. The following table lists the seven types of constraints:

Constraint	Description
`where T : struct`	The type argument must be a value type. Any value type except Nullable can be specified. For more information, see Using Nullable Types.
`where T : class`	The type argument must be a reference type. This constraint applies also to any class, interface, delegate, or array type.
`where T : unmanaged`	The type argument must not be a reference type and must not contain any reference type members at any level of nesting.
`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.

Constraint	Description
`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.

Some of the constraints are mutually exclusive. All value types must have an accessible parameterless constructor. The `struct` constraint implies the `new()` constraint and the `new()` constraint cannot be combined with the `struct` constraint. The `unmanaged` constraint implies the `struct` constraint. The `unmanaged` constraint cannot be combined with either the `struct` or `new()` constraints.

Why use constraints

By constraining the type parameter, you increase the number of allowable operations and method calls to those supported by the constraining type and all types in its inheritance hierarchy. When you design generic classes or methods, if you will be performing any operation on the generic members beyond simple assignment or calling any methods not supported by System.Object, you will have to apply constraints to the type parameter. For example, the base class constraint tells the compiler that only objects of this type or derived from this type will be used as type arguments. Once the compiler has this

guarantee, it can allow methods of that type to be called in the generic class. The following code example demonstrates the functionality you can add to the `GenericList<T>` class (in Introduction to Generics) by applying a base class constraint.

C# Copy

```
public class Employee
{
    public Employee(string s, int i) => (Name, ID) = (s, i);
    public string Name { get; set; }
    public int ID { get; set; }
}

public class GenericList<T> where T : Employee
{
    private class Node
    {
        public Node(T t) => (Next, Data) = (null, t);

        public Node Next { get; set; }
        public T Data { get; set; }
    }

    private Node head;

    public void AddHead(T t)
    {
```

```
        Node n = new Node(t) { Next = head };
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    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. The constraint specifies that all items of type `T` are guaranteed to be either an `Employee` object or an object that inherits from `Employee`.

Multiple constraints can be applied to the same type parameter, and the constraints themselves can be generic types, as follows:

C# Copy

```
class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>, new()
{
    // ...
}
```

When applying the `where T : class` constraint, avoid the `==` and `!=` operators on the type parameter because these operators will test for reference identity only, not for value equality. This

behavior occurs even if these operators are overloaded in a type that is used as an argument. The following code illustrates this point; the output is false even though the String class overloads the `==` operator.

C# Copy

```
public static void OpEqualsTest<T>(T s, T t) where T : class
{
    System.Console.WriteLine(s == t);
}
private static void TestStringEquality()
{
    string s1 = "target";
    System.Text.StringBuilder sb = new System.Text.StringBuilder("target");
    string s2 = sb.ToString();
    OpEqualsTest<string>(s1, s2);
}
```

The compiler only knows that T is a reference type at compile time and must use the default operators that are valid for all reference types. If you must test for value equality, the recommended way is to also apply the `where T : IEquatable<T>` or `where T : IComparable<T>` constraint and implement the interface in any class that will be used to construct the generic class.

Constraining multiple parameters

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

C# Copy

```
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 cannot be used because there is 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 example:

C# Copy

```
public class List<T>
{
    public 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.

Type parameters can also be used as constraints in generic class definitions. The type parameter must be declared within the angle brackets together with any other type parameters:

C# Copy

```
//Type parameter V is used as a type constraint.
public class SampleClass<T, U, V> where T : V { }
```

The usefulness of type parameters as constraints with generic classes is limited because the compiler can assume nothing about the type parameter except that it derives from `System.Object`. Use type parameters as constraints on generic classes in scenarios in which you want to enforce an inheritance relationship between two type parameters.

Unmanaged constraint

Beginning with C# 7.3, you can use the `unmanaged` constraint to specify that the type parameter must be an **unmanaged type**. An **unmanaged type** is a type that is not a reference type and doesn't contain reference type fields at any level of nesting. 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 example:

C# Copy

```
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;
}
```

The preceding method must be compiled in an `unsafe` context because it uses the `sizeof` operator on a type not known to be a built-in type. Without the `unmanaged` constraint, the `sizeof` operator is unavailable.

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 are the same type:

C# Copy

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

You can use the above method to combine delegates that are the same type:

C# Copy

```
Action first = () => Console.WriteLine("this");
Action second = () => Console.WriteLine("that");

var combined = first.TypeSafeCombine(second);
combined();

Func<bool> test = () => true;
// Combine signature ensures combined delegates must
// have the same type.
//var badCombined = first.TypeSafeCombine(test);
```

If you uncomment the last line, it won't compile. Both `first` and `test` are delegate types, but they are different delegate types.

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.

C# Copy

```
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;
}
```

The methods used make use of reflection, which has performance implications. You can call this method to build a collection that is cached and reused rather than repeating the calls that require reflection.

You could use it as shown in the following sample to create an enum and build a dictionary of its values and names:

C# Copy

```
enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
```

```
    Violet
}
```

C# Copy

```
var map = EnumNamedValues<Rainbow>();

foreach (var pair in map)
    Console.WriteLine($"{pair.Key}:\t{pair.Value}");
```

See also

System.Collections.Generic C# Programming Guide

Introduction to Generics

Generic Classes

new Constraint