Constraints on type parameters (C# Programming Guide)

Article • 11/15/2022

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. For more information, see Why use constraints. 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:

Constraint	Description
`where T : struct`	The type argument must be a non-nullable value type. For information about nullable value types, see Nullable value types. 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.
`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, `T` must be a non-nullable reference type.
`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.
`where T : notnull`	The type argument must be a non-nullable type. The argument can be a non-nullable reference type or a non-nullable value type.
`where T : default`	This constraint resolves the ambiguity when you need to specify an unconstrained type parameter when you override a method or provide an explicit interface implementation. The `default` constraint implies the base method without either the `class` or `struct` constraint. For more information, see the default constraint spec proposal.
`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.
`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. The `new()` constraint can't be combined with the `struct` and `unmanaged` constraints.

Constraint	Description
`where T :` *<base class name>*	The type argument must be or derive from the specified base class. In a nullable context, `T` must be a non-nullable reference type derived from the specified base class.
`where T :` *<base class name>?*	The type argument must be or derive from the specified base class. In a nullable context, `T` may be either a nullable or non-nullable type derived 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. In a nullable context, `T` must be a non-nullable type that implements the specified interface.
`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, `T` may be a nullable reference type, a non-nullable reference type, or a value type. `T` may not be a nullable value type.
`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.

Why use constraints

Constraints specify the capabilities and expectations of a type parameter. Declaring those constraints means you can use the operations and method calls of the constraining type. If your generic class or method uses any operation on the generic members beyond simple assignment or calling any methods not supported by System.Object, you'll 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#

```
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
{
    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#

```
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#

```
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#

```
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 example:

C#

```
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#
```

```
//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.

`notnull` constraint

You can use the `notnull` constraint to specify that the type argument must be a non-nullable value type or non-nullable reference type. Unlike most other constraints, if a type argument violates the `notnull` constraint, the compiler generates a warning instead of an error.

The `notnull` constraint has an effect only when used in a nullable context. If you add the `notnull` constraint in a nullable oblivious context, the compiler doesn't generate any warnings or errors for violations of the constraint.

`class` constraint

The `class` constraint in a nullable context specifies that the type argument must be a non-nullable reference type. In a nullable context, when a type argument is a nullable reference type, the compiler generates a warning.

`default` constraint

The addition of nullable reference types complicates the use of `T?` in a generic type or method. `T?` can be used with either the `struct` or `class` constraint, but one of them must be present. When the `class` constraint was used, `T?` referred to the nullable reference type for `T`. Beginning with C# 9, `T?` can be used when neither constraint is applied. In that case, `T?` is interpreted as `T?` for value types and reference types. However, if `T` is an instance of Nullable<T>, `T?` is the same as `T`. In other words, it doesn't become `T??`.

Because `T?` can now be used without either the `class` or `struct` constraint, ambiguities can arise in overrides or explicit interface implementations. In both those cases, the override doesn't include the constraints, but inherits them from the base class. When the base class doesn't apply either the `class` or `struct` constraint, derived classes need to somehow specify an override applies to the base method without either constraint. That's when the derived method applies the `default` constraint. The `default` constraint clarifies *neither* the `class` nor `struct` constraint.

Unmanaged constraint

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 example:

C#

```
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.

The `unmanaged` constraint implies the `struct` constraint and can't be combined with it. Because the `struct` constraint implies the `new()` constraint, the `unmanaged` constraint can't be combined with the `new()` constraint as well.

Delegate constraints

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:

C#

```
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#

```
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're different delegate types.

Enum constraints

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#

```
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;
}
```

`Enum.GetValues` and `Enum.GetName` use reflection, which has performance implications. You can call `EnumNamedValues` 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#

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

C#

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

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

Type arguments implement declared interface

Some scenarios require that an argument supplied for a type parameter implement that interface. For example:

C#

```
public interface IAdditionSubtraction<T> where T :
IAdditionSubtraction<T>
{
    public abstract static T operator +(T left, T right);
    public abstract static T operator -(T left, T right);
}
```

This pattern enables the C# compiler to determine the containing type for the overloaded operators, or any `static virtual` or `static abstract` method. It provides the syntax so that the addition and subtraction operators can be defined on a containing type. Without this constraint, the parameters and arguments would be required to be declared as the interface, rather than the type parameter:

C#

```
public interface IAdditionSubtraction<T> where T :
IAdditionSubtraction<T>
{
    public abstract static IAdditionSubtraction<T> operator +(
        IAdditionSubtraction<T> left,
        IAdditionSubtraction<T> right);

    public abstract static IAdditionSubtraction<T> operator -(
        IAdditionSubtraction<T> left,
        IAdditionSubtraction<T> right);
}
```

The preceding syntax would require implementers to use explicit interface implementation for those methods. Providing the extra constraint enables the interface to define the operators in terms of the type parameters. Types that implement the interface can implicitly implement the interface methods.

See also

- System.Collections.Generic
- C# Programming Guide
- Introduction to Generics
- Generic Classes
- new Constraint