where (generic type constraint) (C# Reference)

Article • 09/29/2022

The `where` clause in a generic definition specifies constraints on the types that are used as arguments for type parameters in a generic type, method, delegate, or local function. Constraints can specify interfaces, base classes, or require a generic type to be a reference, value, or unmanaged type. They declare capabilities that the type argument must have, and must be placed after any declared base class or implemented interfaces.

For example, you can declare a generic class, `AGenericClass`, such that the type parameter `T` implements the IComparable<T> interface:

C#

```
public class AGenericClass<T> where T : IComparable<T> { }
```

> **Note**
>
> For more information on the where clause in a query expression, see **where clause**.

The `where` clause can also include a base class constraint. The base class constraint states that a type to be used as a type argument for that generic type has the specified class as a base class, or is that base class. If the base class constraint is used, it must appear before any other constraints on that type parameter. Some types are disallowed as a base class constraint: Object, Array, and ValueType. The following example shows the types that can now be specified as a base class:

C#

```
public class UsingEnum<T> where T : System.Enum { }

public class UsingDelegate<T> where T : System.Delegate { }

public class Multicaster<T> where T : System.MulticastDelegate { }
```

In a nullable context, the nullability of the base class type is enforced. If the base class is non-nullable (for example `Base`), the type argument must be non-nullable. If the base class is nullable (for example `Base?`), the type argument may be either a nullable or

non-nullable reference type. The compiler issues a warning if the type argument is a nullable reference type when the base class is non-nullable.

The `where` clause can specify that the type is a `class` or a `struct`. The `struct` constraint removes the need to specify a base class constraint of `System.ValueType`. The `System.ValueType` type may not be used as a base class constraint. The following example shows both the `class` and `struct` constraints:

C#

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

In a nullable context, the `class` constraint requires a type to be a non-nullable reference type. To allow nullable reference types, use the `class?` constraint, which allows both nullable and non-nullable reference types.

The `where` clause may include the `notnull` constraint. The `notnull` constraint limits the type parameter to non-nullable types. The type may be a value type or a non-nullable reference type. The `notnull` constraint is available for code compiled in a nullable enable context. Unlike other constraints, if a type argument violates the `notnull` constraint, the compiler generates a warning instead of an error. Warnings are only generated in a `nullable enable` context.

The addition of nullable reference types introduces a potential ambiguity in the meaning of `T?` in generic methods. If `T` is a `struct`, `T?` is the same as System.Nullable<T>. However, if `T` is a reference type, `T?` means that `null` is a valid value. The ambiguity arises because overriding methods can't include constraints. The new `default` constraint resolves this ambiguity. You'll add it when a base class or interface declares two overloads of a method, one that specifies the `struct` constraint, and one that doesn't have either the `struct` or `class` constraint applied:

C#

```
public abstract class B
{
    public void M<T>(T? item) where T : struct { }
    public abstract void M<T>(T? item);

}
```

You use the `default` constraint to specify that your derived class overrides the method without the constraint in your derived class, or explicit interface implementation. It's only valid on methods that override base methods, or explicit interface implementations:

C#

```
public class D : B
{
    // Without the "default" constraint, the compiler tries to over-
ride the first method in B
    public override void M<T>(T? item) where T : default { }
}
```

> **Important**
>
> Generic declarations that include the `notnull` constraint can be used in a nullable oblivious context, but compiler does not enforce the constraint.

C#

```
#nullable enable
    class NotNullContainer<T>
        where T : notnull
    {
    }
#nullable restore
```

The `where` clause may also include an `unmanaged` constraint. The `unmanaged` constraint limits the type parameter to types known as unmanaged types. The `unmanaged` constraint makes it easier to write low-level interop code in C#. This constraint enables reusable routines across all unmanaged types. The `unmanaged` constraint can't be combined with the `class` or `struct` constraint. The `unmanaged` constraint enforces that the type must be a `struct`:

C#

```
class UnManagedWrapper<T>
    where T : unmanaged
{ }
```

The `where` clause may also include a constructor constraint, `new()`. That constraint makes it possible to create an instance of a type parameter using the `new` operator. The

new() Constraint lets the compiler know that any type argument supplied must have an accessible parameterless constructor. For example:

C#

```
public class MyGenericClass<T> where T : IComparable<T>, new()
{
    // The following line is not possible without new() constraint:
    T item = new T();
}
```

The `new()` constraint appears last in the `where` clause. The `new()` constraint can't be combined with the `struct` or `unmanaged` constraints. All types satisfying those constraints must have an accessible parameterless constructor, making the `new()` constraint redundant.

With multiple type parameters, use one `where` clause for each type parameter, for example:

C#

```
public interface IMyInterface { }

namespace CodeExample
{
    class Dictionary<TKey, TVal>
        where TKey : IComparable<TKey>
        where TVal : IMyInterface
    {
        public void Add(TKey key, TVal val) { }
    }
}
```

You can also attach constraints to type parameters of generic methods, as shown in the following example:

C#

```
public void MyMethod<T>(T t) where T : IMyInterface { }
```

Notice that the syntax to describe type parameter constraints on delegates is the same as that of methods:

C#

```
delegate T MyDelegate<T>() where T : new();
```

For information on generic delegates, see Generic Delegates.

For details on the syntax and use of constraints, see Constraints on Type Parameters.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- Introduction to Generics
- new Constraint
- Constraints on Type Parameters