**Generic Constraints in C#**

Constraints in C# are nothing but validations that we can put on the generic type parameter. That means constraints are used to restrict the types that can be substituted for type parameters. It will give you a compile-time error if you try to substitute a generic type using a type that is not allowed by the specified constraints. It is also possible to specify one or more constraints on the generic type using the where clause after the generic type name. If this is not clear at the moment, don’t worry we will try to understand this with examples.

**Why do we need Generic Constraints in C#?**

Let us first understand why we need constraints and then we will see the different types of generic constraints in C# with examples. As we already discussed generics are used to define a class or structure or method with placeholders (type parameters) to indicate that they can use any of the types. For a better understanding, please have a look at the below example. The following example shows a generic class with type parameter (T) as a placeholder with angle (<>) brackets.

**using** *System;*

**namespace** *GenericsConstraintsDemo*

**{**

**public** **class** GenericClass**<**T**>**

**{**

**public** T Message;

**public** **void** GenericMethod**(**T Param1, T Param2**)**

**{**

Console.WriteLine**(**$"Message: {Message}"**)**;

Console.WriteLine**(**$"Param1: {Param1}"**)**;

Console.WriteLine**(**$"Param2: {Param2}"**)**;

**}**

**}**

**}**

As you can see in the above code, here we have created a class i.e. GenericClass with one variable i.e. Message and one method i.e. GenericMethod using type parameter T and we have defined this type parameter with the class name using the angle (<>) brackets. Here, the GenericClass doesn’t know anything about the defined placeholder i.e. T, and hence it will accept any type of value i.e. it can accept a string, int, struct, boolean, class, etc., based on our requirements. This is the reason why we are using generics in C#.

But, if you want to restrict a generic class to accept only a particular type of placeholder, then we need to use the Generic Constraints. So, by using Generic Constraints in C#, we can specify what type of placeholder the generic class can accept. If we try to instantiate a generic class with the placeholder type, that is not allowed by a constraint, then the compiler will throw a compile-time error. For example, if we specify the generic type to accept on class type, then later if we try to send int, bool, or any value type, then we will get a compile-time error. Now, I hope you understand why we need Generic Constraints in C#.

**Syntax: GenericTypeName<T> where T : contraint1, constraint2**

**Types of Generic Constraints in C#:**

Constraints are validations that we can put on the generic type parameters. At the instantiation time of the generic class, if we provide an invalid type, then the compile will give an error. In C#, the generic constraints are specified by using the where keyword. The following are the list of different type of generic constraints available in c#.

1. **where T: struct** => The type argument must be non-nullable value types such as primitive data types int, double, char, bool, float, etc. The struct constraint can’t be combined with the unmanaged constraint.
2. **where T: class** => The type argument must be a reference type. This constraint can be applied to any class (non-nullable), interface, delegate, or array type in C#.
3. **where T: new()** => The type argument must be a reference type that has a public parameterless (default) constructor.
4. **where T: <base class name>** => The type of argument must be or derive from the specified base class.
5. **where T: <interface name>** => The type argument must be or implement the specified interface. Also, multiple interface constraints can be specified.
6. **where T: U** => The type argument supplied for 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 a non-nullable reference type. If U is a nullable reference type, T may be either nullable or non-nullable.

Now, let us proceed further and understand the use of each constraint in generics with examples.

**where T: class Generic Constraint in C#**

The type argument must be a reference type. We know a class is a reference type in C#. So “**where T: class**” is a reference type constraint. That means this constraint can be applied to any class (non-nullable), interface, delegate, or array type in C#. For a better understanding, please have a look at the below example.

**using** *System;*

**namespace** *GenericsConstraintsDemo*

**{**

**public** **class** GenericClass**<**T**>** **where** T : **class**

**{**

**public** T Message;

**public** **void** GenericMethod**(**T Param1, T Param2**)**

**{**

Console.WriteLine**(**$"Message: {Message}"**)**;

Console.WriteLine**(**$"Param1: {Param1}"**)**;

Console.WriteLine**(**$"Param2: {Param2}"**)**;

**}**

**}**

**}**

If you observe the above code, here, we defined the GenericClass with the “**where T: class**” constraint. That means, now the GenericClass will only accept reference-type arguments. Let’s create an instance of the Generic class by passing reference-type arguments as follows. In C#, the string is a reference type.  
**GenericClass<string> stringClass = new GenericClass<string>();**

The following statement will give you a compile-time error as int is a value type, not a reference type.  
**GenericClass<int> intClass = new GenericClass<int>();**

**Example to understand where T: class Generic Constraint in C#**

When we created an instance of GenericClass using reference type arguments such as string and class, it works fine. But, when we try to create an instance with built-in types like int, bool, etc., we will get a compile-time error.

**using** *System;*

**namespace** *GenericsConstraintsDemo*

**{**

**public** **class** GenericClass**<**T**>** **where** T : **class**

**{**

**public** T Message;

**public** **void** GenericMethod**(**T Param1, T Param2**)**

**{**

Console.WriteLine**(**$"Message: {Message}"**)**;

Console.WriteLine**(**$"Param1: {Param1}"**)**;

Console.WriteLine**(**$"Param2: {Param2}"**)**;

**}**

**}**

**public** **class** Employee

**{**

**public** **string** Name **{** **get**; **set**; **}**

**public** **string** Location **{** **get**; **set**; **}**

**}**

**class** Program

**{**

**static** **void** Main**()**

**{**

// Instantiate Generic Class with Constraint

GenericClass**<string>** stringClass = new GenericClass**<string>()**;

stringClass.Message = "Welcome to DotNetTutorials";

stringClass.GenericMethod**(**"Anurag Mohanty", "Bhubaneswar"**)**;

GenericClass**<**Employee**>** EmployeeClass = new GenericClass**<**Employee**>()**;

Employee emp1 = new Employee**()** **{** Name = "Anurag", Location = "Bhubaneswar" **}**;

Employee emp2 = new Employee**()** **{** Name = "Mohanty", Location = "Cuttack" **}**;

Employee emp3 = new Employee**()** **{** Name = "Sambit", Location = "Delhi" **}**;

EmployeeClass.Message = emp1;

EmployeeClass.GenericMethod**(**emp2, emp3**)**;

// The following statement will giveCompile Time Error as int is a value type, not reference type

//GenericClass<int> intClass = new GenericClass<int>();

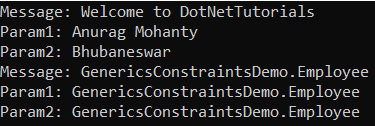
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**where T: struct Generic Constraint in C#**

If you want the type argument to accept only the value type then you need to use where T: struct constraints in C#. In this case, the type argument must be non-nullable value types such as int, double, char, bool, float, etc. The struct constraint can’t be combined with the unmanaged constraint. Let us see an example to understand where T: struct constraint. For a better understanding, please have a look at the below example.

**using** *System;*

**namespace** *GenericsConstraintsDemo*

**{**

**public** **class** GenericClass**<**T**>** **where** T : struct

**{**

**public** T Message;

**public** **void** GenericMethod**(**T Param1, T Param2**)**

**{**

Console.WriteLine**(**$"Message: {Message}"**)**;

Console.WriteLine**(**$"Param1: {Param1}"**)**;

Console.WriteLine**(**$"Param2: {Param2}"**)**;

**}**

**}**

**}**

If you observe the above code, here, we defined the GenericClass with the “**where T: stru**ct” generic constraint. That means, now the GenericClass will only accept value-type arguments. Let us create an instance of GenericClass by passing value-type arguments as follows.  
**GenericClass<int> intClass = new GenericClass<int>();**

The following statement will give you a compile-time error as the string is a reference type, not a value type.  
**GenericClass<string> stringClass = new GenericClass<string>();**

**Example to understand where T: struct Generics Constraint in C#**

When we created an instance of GenericClass using value-type arguments such as int, it works fine. But, when we try to create an instance with reference types such as String, Employee, etc., we will get a compile-time error.

**using** *System;*

**namespace** *GenericsConstraintsDemo*

**{**

**public** **class** GenericClass**<**T**>** **where** T : struct

**{**

**public** T Message;

**public** **void** GenericMethod**(**T Param1, T Param2**)**

**{**

Console.WriteLine**(**$"Message: {Message}"**)**;

Console.WriteLine**(**$"Param1: {Param1}"**)**;

Console.WriteLine**(**$"Param2: {Param2}"**)**;

**}**

**}**

**public** **class** Employee

**{**

**public** **string** Name **{** **get**; **set**; **}**

**public** **string** Location **{** **get**; **set**; **}**

**}**

**public** **class** Program

**{**

**static** **void** Main**()**

**{**

// Instantiate Generic Class with Constraint

GenericClass**<int>** intClass = new GenericClass**<int>()**;

intClass.Message = 30;

intClass.GenericMethod**(**10, 20**)**;

//Compile Time Error as string is not a value type, it is a reference type

//GenericClass<string> stringClass = new GenericClass<string>();

//Compile Time Error as Employee is not a value type, it is a reference type

//GenericClass<Employee> EmployeeClass = new GenericClass<Employee>();

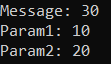
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**where T: new() Generic Constraint in C#**

Here, the type argument must be a reference type that has a public parameterless (default) constructor. That means with the help of the new() constraint, we can only specify types which has a parameterless constructor. For a better understanding, please have a look at the below example.

**using** *System;*

**namespace** *GenericsDemo*

**{**

**public** **class** GenericClass**<**T**>** **where** T : new**()**

**{**

**public** T Message;

**public** **void** GenericMethod**(**T Param1, T Param2**)**

**{**

Console.WriteLine**(**$"Message: {Message}"**)**;

Console.WriteLine**(**$"Param1: {Param1}"**)**;

Console.WriteLine**(**$"Param2: {Param2}"**)**;

**}**

**}**

**}**

As you can see in the above code, we have used **where T: new()** constraint which allows the type which has a parameterless default constructor. Now, let us create two more classes with one class having a parameterless default constructor and another class having parameterized constructor as follows.

**namespace** *GenericsDemo*

**{**

**public** **class** Employee

**{**

**public** **string** Name **{** **get**; **set**; **}**

**public** **string** Location **{** **get**; **set**; **}**

**}**

**public** **class** Customer

**{**

**public** **string** Name **{** **get**; **set**; **}**

**public** **string** Location **{** **get**; **set**; **}**

**public** Customer**(string** customerName, **string** customerLocation**)**

**{**

Name = customerName;

Location = customerLocation;

**}**

**}**

**}**

As you can see in the above code, we have not defined any constructor explicitly in the Employee class, so the compiler will provide a parameter-less default constructor. On the other hand, in the Customer class, we have defined one parameterized constructor explicitly. Now, Let us create an instance of GenericClass bypassing Employee type arguments as follows.  
**GenericClass<Employee> employee = new GenericClass<Employee>();**

The following statement will give you a compile-time error as the Customer class has a Parameterized constructor or you can say the Customer class does not have a default parameterless constructor.  
**GenericClass<Customer> customer = new GenericClass<Customer>();**

**Example to understand where T: new() Generic Constraint in C#**

When we created an instance of GenericClass using the Employee type argument, it works fine. But, when we try to create an instance with Customer type, we will get a compile-time error.

**using** *System;*

**namespace** *GenericsDemo*

**{**

**public** **class** GenericClass**<**T**>** **where** T : new**()**

**{**

**public** T Message;

**public** **void** GenericMethod**(**T Param1, T Param2**)**

**{**

Console.WriteLine**(**$"Message: {Message}"**)**;

Console.WriteLine**(**$"Param1: {Param1}"**)**;

Console.WriteLine**(**$"Param2: {Param2}"**)**;

**}**

**}**

**public** **class** Employee

**{**

**public** **string** Name **{** **get**; **set**; **}**

**public** **string** Location **{** **get**; **set**; **}**

**}**

**public** **class** Customer

**{**

**public** **string** Name **{** **get**; **set**; **}**

**public** **string** Location **{** **get**; **set**; **}**

**public** Customer**(string** customerName, **string** customerLocation**)**

**{**

Name = customerName;

Location = customerLocation;

**}**

**}**

**class** Program

**{**

**static** **void** Main**()**

**{**

//No Error, as Emplyoee class has parameterless constructor

GenericClass**<**Employee**>** employee = new GenericClass**<**Employee**>()**;

Employee emp1 = new Employee**()** **{** Name = "Anurag", Location = "Bhubaneswar" **}**;

Employee emp2 = new Employee**()** **{** Name = "Mohanty", Location = "Cuttack" **}**;

Employee emp3 = new Employee**()** **{** Name = "Sambit", Location = "Delhi" **}**;

employee.Message = emp1;

employee.GenericMethod**(**emp2, emp3**)**;

//CompileTime Error, as Customer class has Parameterized constructor

//GenericClass<Customer> customer = new GenericClass<Customer>();

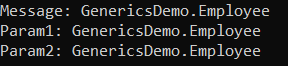
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**where T: BaseClass Generic Constraint in C#**

Here, the type of argument must be derived from the specified base class. That means in the <base class> constraint, we can only specify types that are inherited from <base class>. The following example shows the base class constraint that restricts the type argument to be a derived class of the specified class. For a better understanding, please have a look at the below example.

**using** *System;*

**namespace** *GenericsDemo*

**{**

**public** **class** GenericClass**<**T**>** **where** T : BaseEmployee

**{**

**public** T Message;

**public** **void** GenericMethod**(**T Param1, T Param2**)**

**{**

Console.WriteLine**(**$"Message: {Message}"**)**;

Console.WriteLine**(**$"Param1: {Param1}"**)**;

Console.WriteLine**(**$"Param2: {Param2}"**)**;

**}**

**}**

**}**

As you can see in the above code, here we have uses **where T: BaseEmployee** constraint which allows the type which is the derived class, abstract class, and interface of the BaseEmployee type. Now, let us create three more classes as follows.

**namespace** *GenericsDemo*

**{**

**public** **class** BaseEmployee

**{**

**}**

**public** **class** Employee : BaseEmployee

**{**

**public** **string** Name **{** **get**; **set**; **}**

**}**

**public** **class** Customer

**{**

**public** **string** Name **{** **get**; **set**; **}**

**}**

**}**

As you can see in the above code, the Employee class is inherited from the BaseEmployee class i.e. Employee is the derived class of the BaseEmployee class. On the other hand, Customer is not derived from the BaseEmployee class. Now, Let’s create an instance of GenericClass bypassing Employee type arguments as follows. It works fine because Employee is derived from the BaseEmployee class.  
**GenericClass<Employee> employee = new GenericClass<Employee>();**

The following statement will give you a compile-time error as the Customer class is not derived from the BaseEmployee type.  
**GenericClass<Customer> customer = new GenericClass<Customer>();**

**Example to understand where T: BaseClass Constraint in C# Generics**

When we created an instance of GenericClass using the Employee type argument, it works fine because Employee is the derived class of the BaseEmployee class. But, when we try to create an instance with Customer type, we will get a compile-time error because Customer is not a derived class of the BaseEmployee class.

**using** *System;*

**namespace** *GenericsDemo*

**{**

**public** **class** GenericClass**<**T**>** **where** T : BaseEmployee

**{**

**public** T Message;

**public** **void** GenericMethod**(**T Param1, T Param2**)**

**{**

Console.WriteLine**(**$"Message: {Message}"**)**;

Console.WriteLine**(**$"Param1: {Param1}"**)**;

Console.WriteLine**(**$"Param2: {Param2}"**)**;

**}**

**}**

**public** **class** BaseEmployee

**{**

**}**

**public** **class** Employee : BaseEmployee

**{**

**public** **string** Name **{** **get**; **set**; **}**

**}**

**public** **class** Customer

**{**

**public** **string** Name **{** **get**; **set**; **}**

**}**

**class** Program

**{**

**static** **void** Main**()**

**{**

//No Error, as Emplyoee is a derived class of BaseEmployee class

GenericClass**<**Employee**>** employee = new GenericClass**<**Employee**>()**;

Employee emp1 = new Employee**()** **{** Name = "Anurag" **}**;

Employee emp2 = new Employee**()** **{** Name = "Mohanty" **}**;

Employee emp3 = new Employee**()** **{** Name = "Sambit" **}**;

employee.Message = emp1;

employee.GenericMethod**(**emp2, emp3**)**;

//CompileTime Error, as Customer is not a derived class of BaseEmployee class

//GenericClass<Customer> customer = new GenericClass<Customer>();

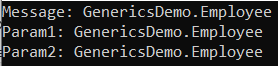
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**where T: Interface Generic Constraint in C#**

Here, the type argument must be or implement the specified interface. Also, multiple interface constraints can be specified. That means in the <interface> constraint, we can only specify types that implement the <interface>. For a better understanding, please have a look at the below example.

**using** *System;*

**namespace** *GenericsDemo*

**{**

**public** **class** GenericClass**<**T**>** **where** T : IEmployee

**{**

**public** T Message;

**public** **void** GenericMethod**(**T Param1, T Param2**)**

**{**

Console.WriteLine**(**$"Message: {Message}"**)**;

Console.WriteLine**(**$"Param1: {Param1}"**)**;

Console.WriteLine**(**$"Param2: {Param2}"**)**;

**}**

**}**

**}**

As you can see in the above code, here we have uses **where T: IEmployee** constraint which allows the type that should implement the IEmployee interface. Now, let us create one interface and two more classes as follows.

**namespace** *GenericsDemo*

**{**

**public** **interface** IEmployee

**{**

**}**

**public** **class** Employee : IEmployee

**{**

**public** **string** Name **{** **get**; **set**; **}**

**}**

**public** **class** Customer

**{**

**public** **string** Name **{** **get**; **set**; **}**

**}**

**}**

As you can see in the above code, the Employee class is implemented IEmployee interface. On the other hand, the Customer is not implementing the IEmployee interface. Now, let us create an instance of GenericClass bypassing Employee type arguments as follows. It works fine because the Employee class implements the IEmployee interface.  
**GenericClass<Employee> employee = new GenericClass<Employee>();**

The following statement will give you a compile-time error as the Customer class does not implement the IEmployee interface.  
**GenericClass<Customer> customer = new GenericClass<Customer>();**

**Example to understand where T: Interface Constraint in C# Generics**

When we created an instance of GenericClass using the Employee type argument, it works fine because the Employee class implements the IEmployee interface. But, when we try to create an instance with Customer type, we will get a compile-time error because the Customer class does not implement the IEmployee interface.

**using** *System;*

**namespace** *GenericsDemo*

**{**

**public** **class** GenericClass**<**T**>** **where** T : IEmployee

**{**

**public** T Message;

**public** **void** GenericMethod**(**T Param1, T Param2**)**

**{**

Console.WriteLine**(**$"Message: {Message}"**)**;

Console.WriteLine**(**$"Param1: {Param1}"**)**;

Console.WriteLine**(**$"Param2: {Param2}"**)**;

**}**

**}**

**public** **interface** IEmployee

**{**

**}**

**public** **class** Employee : IEmployee

**{**

**public** **string** Name **{** **get**; **set**; **}**

**}**

**public** **class** Customer

**{**

**public** **string** Name **{** **get**; **set**; **}**

**}**

**class** Program

**{**

**static** **void** Main**()**

**{**

//No Error, as Emplyoee class Implement the IEmployee Interface

GenericClass**<**Employee**>** employee = new GenericClass**<**Employee**>()**;

Employee emp1 = new Employee**()** **{** Name = "Anurag" **}**;

Employee emp2 = new Employee**()** **{** Name = "Mohanty" **}**;

Employee emp3 = new Employee**()** **{** Name = "Sambit" **}**;

employee.Message = emp1;

employee.GenericMethod**(**emp2, emp3**)**;

//CompileTime Error, as Customer is not Implement the IEmployee Interface

//GenericClass<Customer> customer = new GenericClass<Customer>();

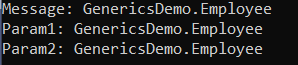
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**where T: U Generic Constraint in C#**

Here, the type argument supplied 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 a non-nullable reference type. If U is a nullable reference type, T may be either nullable or non-nullable. So, in this constraint, there are two Type Arguments i.e. T and U. U can be an interface, abstract class, or simple class. T must inherit or implements the U class. For a better understanding, please have a look at the below code.

**using** *System;*

**namespace** *GenericsDemo*

**{**

**public** **class** GenericClass**<**T, U**>** **where** T : U

**{**

**public** T Message;

**public** **void** GenericMethod**(**T Param1, U Param2**)**

**{**

Console.WriteLine**(**$"Message: {Message}"**)**;

Console.WriteLine**(**$"Param1: {Param1}"**)**;

Console.WriteLine**(**$"Param2: {Param2}"**)**;

**}**

**}**

**}**

As you can see in the above code, here we have used **where T: U** constraint which allows the type (T) that must inherit or implements the U class. Now, let us create one interface and two more classes as follows.

**namespace** *GenericsDemo*

**{**

**public** **interface** IEmployee

**{**

**}**

**public** **class** Employee : IEmployee

**{**

**public** **string** Name **{** **get**; **set**; **}**

**}**

**public** **class** Customer

**{**

**public** **string** Name **{** **get**; **set**; **}**

**}**

**}**

As you can see in the above code, the Employee class implements the IEmployee interface. On the other hand, the Customer class is not implementing the IEmployee interface. Now, Let’s create an instance of Genericclass bypassing Employee and IEmployee as type arguments for T and U as follows. It works fine because the Employee class implements the IEmployee interface.  
**GenericClass<Employee, IEmployee> employeeGenericClass = new GenericClass<Employee, IEmployee>();**

The following statement will give you a compile-time error as the Customer class does not implement the IEmployee interface i.e. T does not implement U.  
**GenericClass<Customer, IEmployee> customerGenericClass = new GenericClass<Customer, IEmployee>();**

**Example to understand where T: U constraint in C# generics**

When we created an instance of GenericClass using Employee and IEmployee type argument, it works fine because the Employee class implements the IEmployee interface. But, when we try to create an instance with Customer type, we will get a compile-time error because the Customer class does not implement the IEmployee interface.

**using** *System;*

**namespace** *GenericsDemo*

**{**

**public** **class** GenericClass**<**T, U**>** **where** T : U

**{**

**public** T Message;

**public** **void** GenericMethod**(**T Param1, U Param2**)**

**{**

Console.WriteLine**(**$"Message: {Message}"**)**;

Console.WriteLine**(**$"Param1: {Param1}"**)**;

Console.WriteLine**(**$"Param2: {Param2}"**)**;

**}**

**}**

**public** **interface** IEmployee

**{**

**}**

**public** **class** Employee : IEmployee

**{**

**public** **string** Name **{** **get**; **set**; **}**

**}**

**public** **class** Customer

**{**

**public** **string** Name **{** **get**; **set**; **}**

**}**

**class** Program

**{**

**static** **void** Main**()**

**{**

//No Error, as Emplyoee class Implement the IEmployee Interface i.e. T Implements U

GenericClass**<**Employee, IEmployee**>** employeeGenericClass = new GenericClass**<**Employee, IEmployee**>()**;

//CompileTime Error, as Customer is not Implement the IEmployee Interface i.e. T does not Implements U

// GenericClass<Customer, IEmployee> customerGenericClass = new GenericClass<Customer, IEmployee>();

Console.ReadKey**()**;

**}**

**}**

**}**

**Multiple Generic Constraints in C#:**

In C# generics, it is also possible to apply multiple constraints on generic classes based on our requirements. Let us understand this with an example. In the below example, we are creating the generic class with two constraints. The first constraint specifies that the T parameter must be a reference type whereas the second constraint specifies that the X parameter must be a value type.

**using** *System;*

**namespace** *GenericsDemo*

**{**

**public** **class** GenericClass**<**T, X**>** **where** T: **class** **where** X: struct

**{**

**public** T Message;

**public** **void** GenericMethod**(**T Param1, X Param2**)**

**{**

Console.WriteLine**(**$"Message: {Message}"**)**;

Console.WriteLine**(**$"Param1: {Param1}"**)**;

Console.WriteLine**(**$"Param2: {Param2}"**)**;

**}**

**}**

**class** Program

**{**

**static** **void** Main**()**

**{**

GenericClass**<string**, **int>** multipleGenericConstraints = new GenericClass**<string**, **int>()**;

multipleGenericConstraints.Message = "Good Morning";

multipleGenericConstraints.GenericMethod**(**"Anurag", 100**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

