**Generics**

At its core, the term generics means parameterized types. Parameterized types are important

because they enable you to create classes, structures, interfaces, methods, and delegates in

which the type of data upon which they operate is specified as a parameter. Using generics,

it is possible to create a single class, for example, that automatically works with different

types of data. A class, structure, interface, method, or delegate that operates on a parameterized

type is called generic, as in generic class or generic method.

It is important to understand that C# has always given you the ability to create generalized

code by operating through references of type object. Because object is the base class of all

other classes, an object reference can refer to any type of object. Thus, in pre-generics code,

generalized code used object references to operate on a variety of different kinds of objects.

**Some Generic Restrictions**

Here are a few restrictions that you need to keep in mind when using generics:

* Properties, operators, and indexers cannot be generic. However, these items can be

used in a generic class and can make use of the generic type parameters of that class.

* The extern modifier cannot be applied to a generic method.
* Pointer types cannot be used as type arguments.
* If a generic class contains a static field, then each constructed type has its own copy of

that field. This means that each instance of the same constructed type shares the same

static field. However, a different constructed type shares a different copy of that

field. Thus, a static field is not shared by all constructed types.

// Generic

using System;

class G<T>

{

T obj;

public G(T obj)

{

this.obj = obj;

}

public T GetObject()

{

return obj;

}

public void ShowType()

{

Console.WriteLine("Type: " + typeof(T));

}

}

class MainApp

{

static void Main()

{

G<int> iObj = new G<int>(100);

iObj.ShowType();

int v = iObj.GetObject();

Console.WriteLine("Value: " + v);

Console.WriteLine();

G<string> strObj = new G<string>("Generic");

strObj.ShowType();

string str = strObj.GetObject();

Console.WriteLine("Value: " + str);

}

}

/\* Output:

Type: System.Int32

Value: 100

Type: System.String

Value: Generic

\*/

// Non-Generic

using System;

class G

{

object obj;

public G(object obj)

{

this.obj = obj;

}

public object GetObject()

{

return obj;

}

public void ShowType()

{

Console.WriteLine("Type: " + obj.GetType());

}

}

class MainApp

{

static void Main()

{

G iObj = new G(100);

iObj.ShowType();

int v = (int)iObj.GetObject();

Console.WriteLine("Value: " + v);

Console.WriteLine();

G strObj = new G("Non-Generic");

strObj.ShowType();

string str = (string)strObj.GetObject();

Console.WriteLine("Value: " + str);

}

}

/\* Output:

Type: System.Int32

Value: 100

Type: System.String

Value: Non-Generic

\*/

Visual Studio 2010 - Visual C#

**Constraints on Type Parameters (C# Programming Guide)**

When you define a generic class, you can apply restrictions to the kinds of types that client code can use for type arguments when it instantiates your class. 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. These restrictions are called constraints. Constraints are specified by using the **where** contextual keyword. The following table lists the six types of constraints:

|  |  |
| --- | --- |
| **Constraint** | **Description** |
| where T: struct | The type argument must be a value type. Any value type except [Nullable](http://msdn.microsoft.com/en-us/library/system.nullable.aspx) can be specified. See [Using Nullable Types (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/2cf62fcy.aspx) for more information. |
| where T : class | The type argument must be a reference type; this applies also to any class, interface, delegate, or array type. |
| 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. |
| 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. |

Why Use Constraints

If you want to examine an item in a generic list to determine whether it is valid or to compare it to some other item, the compiler must have some guarantee that the operator or method it has to call will be supported by any type argument that might be specified by client code. This guarantee is obtained by applying one or more constraints to your generic class definition. 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. Constraints are applied by using the contextual keyword **where**. The following code example demonstrates the functionality we can add to the GenericList<T> class (in [Introduction to Generics (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/0x6a29h6.aspx)) by applying a base class constraint.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl24_ctl00_ctl01_code');" \o "Copy Code)

public class Employee

{

private string name;

private int id;

public Employee(string s, int i)

{

name = s;

id = i;

}

public string Name

{

get { return name; }

set { name = value; }

}

public int ID

{

get { return id; }

set { id = value; }

}

}

public class GenericList<T> where T : Employee

{

private class Node

{

private Node next;

private T data;

public Node(T t)

{

next = null;

data = t;

}

public Node Next

{

get { return next; }

set { next = value; }

}

public T Data

{

get { return data; }

set { data = value; }

}

}

private Node head;

public GenericList() //constructor

{

head = null;

}

public void AddHead(T t)

{

Node n = new Node(t);

n.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 because 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 Code](javascript:CopyCode('ctl00_MTCS_main_ctl24_ctl00_ctl02_code');" \o "Copy Code)

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

{

// ...

}

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. Therefore, 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.

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 is the case 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](http://msdn.microsoft.com/en-us/library/system.string.aspx) class overloads the **==** operator.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl24_ctl00_ctl04_code');" \o "Copy Code)

public static void OpTest<T>(T s, T t) where T : class

{

System.Console.WriteLine(s == t);

}

static void Main()

{

string s1 = "target";

System.Text.StringBuilder sb = new System.Text.StringBuilder("target");

string s2 = sb.ToString();

OpTest<string>(s1, s2);

}

The reason for this behavior is that, at compile time, the compiler only knows that T is a reference type, and therefore 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 : IComparable<T> constraint and implement that 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 Code](javascript:CopyCode('ctl00_MTCS_main_ctl25_ctl00_ctl00_code');" \o "Copy Code)

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 to [null](http://msdn.microsoft.com/en-us/library/edakx9da.aspx). 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 Code](javascript:CopyCode('ctl00_MTCS_main_ctl27_ctl00_ctl00_code');" \o "Copy Code)

class List<T>

{

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. Note that the type parameter must be declared within the angle brackets together with any other type parameters:

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl27_ctl00_ctl01_code');" \o "Copy Code)

//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 very 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.