Contents

Generics (C# Programming Guide)

5/4/2018 • 2 minutes to read • Edit Online

Generics were added to version 2.0 of the C# language and the common language runtime (CLR). Generics introduce to the .NET Framework the concept of type parameters, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code. For example, by using a generic type parameter T you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations, as shown here:

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

Generics Overview

- Use generic types to maximize code reuse, type safety, and performance.
- The most common use of generics is to create collection classes.
- The .NET Framework class library contains several new generic collection classes in the System.Collections.Generic namespace. These should be used whenever possible instead of classes such as ArrayList in the System.Collections namespace.
- You can create your own generic interfaces, classes, methods, events and delegates.
- Generic classes may be constrained to enable access to methods on particular data types.
- Information on the types that are used in a generic data type may be obtained at run-time by using reflection.

Related Sections

For more information:

- Introduction to Generics

- Benefits of Generics
- Generic Type Parameters
- Constraints on Type Parameters
- Generic Classes
- Generic Interfaces
- Generic Methods
- Generic Delegates
- Differences Between C++ Templates and C# Generics
- Generics and Reflection
- Generics in the Run Time

C# Language Specification

For more information, see the C# Language Specification.

See Also

System.Collections.Generic
C# Programming Guide
Types
<typeparam>
<typeparamref>
Generics in .NET

Introduction to Generics (C# Programming Guide)

5/4/2018 • 2 minutes to read • Edit Online

Generic classes and methods combine reusability, type safety and efficiency in a way that their non-generic counterparts cannot. Generics are most frequently used with collections and the methods that operate on them. Version 2.0 of the .NET Framework class library provides a new namespace, System.Collections.Generic, which contains several new generic-based collection classes. It is recommended that all applications that target the .NET Framework 2.0 and later use the new generic collection classes instead of the older non-generic counterparts such as ArrayList. For more information, see Generics in .NET.

Of course, you can also create custom generic types and methods to provide your own generalized solutions and design patterns that are type-safe and efficient. The following code example shows a simple generic linked-list class for demonstration purposes. (In most cases, you should use the List<T> class provided by the .NET Framework class library instead of creating your own.) The type parameter `T` is used in several locations where a concrete type would ordinarily be used to indicate the type of the item stored in the list. It is used in the following ways:

- As the type of a method parameter in the `AddHead` method.
- As the return type of the `Data` property in the nested `Node` class.
- As the type of the private member `data` in the nested class.

Note that T is available to the nested `Node` class. When `GenericList<T>` is instantiated with a concrete type, for example as a `GenericList<int>`, each occurrence of `T` will be replaced with `int`.

```
// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node next;
        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }

    private Node head;

    // constructor
    public GenericList()
    {
        head = null;
    }

    // T as method parameter type:
    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;
        }
    }
}
```

The following code example shows how client code uses the generic `GenericList<T>` class to create a list of integers. Simply by changing the type argument, the following code could easily be modified to create lists of strings or any other custom type:

```
class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}
```

See Also

System.Collections.Generic
C# Programming Guide
Generics

Benefits of Generics (C# Programming Guide)

5/4/2018 • 2 minutes to read • Edit Online

Generics provide the solution to a limitation in earlier versions of the common language runtime and the C# language in which generalization is accomplished by casting types to and from the universal base type Object. By creating a generic class, you can create a collection that is type-safe at compile-time.

The limitations of using non-generic collection classes can be demonstrated by writing a short program that uses the ArrayList collection class from the .NET class library. An instance of the ArrayList class can store any reference or value type.

```
// The .NET Framework 1.1 way to create a list:
System.Collections.ArrayList list1 = new System.Collections.ArrayList();
list1.Add(3);
list1.Add(105);

System.Collections.ArrayList list2 = new System.Collections.ArrayList();
list2.Add("It is raining in Redmond.");
list2.Add("It is snowing in the mountains.");
```

But this convenience comes at a cost. Any reference or value type that is added to an ArrayList is implicitly upcast to Object. If the items are value types, they must be boxed when they are added to the list, and unboxed when they are retrieved. Both the casting and the boxing and unboxing operations decrease performance; the effect of boxing and unboxing can be very significant in scenarios where you must iterate over large collections.

The other limitation is lack of compile-time type checking; because an ArrayList casts everything to Object, there is no way at compile-time to prevent client code from doing something such as this:

```
System.Collections.ArrayList list = new System.Collections.ArrayList();
// Add an integer to the list.
list.Add(3);
// Add a string to the list. This will compile, but may cause an error later.
list.Add("It is raining in Redmond.");

int t = 0;
// This causes an InvalidCastException to be returned.
foreach (int x in list)
{
    t += x;
}
```

Although perfectly acceptable and sometimes intentional if you are creating a heterogeneous collection, combining strings and `ints` in a single ArrayList is more likely to be a programming error, and this error will not be detected until runtime.

In versions 1.0 and 1.1 of the C# language, you could avoid the dangers of generalized code in the .NET Framework base class library collection classes only by writing your own type specific collections. Of course, because such a class is not reusable for more than one data type, you lose the benefits of generalization, and you have to rewrite the class for each type that will be stored.

What ArrayList and other similar classes really need is a way for client code to specify, on a per-instance basis, the particular data type that they intend to use. That would eliminate the need for the upcast to Object and would also make it possible for the compiler to do type checking. In other words, ArrayList needs a type parameter. That is exactly what generics provide. In the generic List<T> collection, in the System.Collections.Generic namespace, the

same operation of adding items to the collection resembles this:

```
// The .NET Framework 2.0 way to create a list
List<int> list1 = new List<int>();

// No boxing, no casting:
list1.Add(3);

// Compile-time error:
// list1.Add("It is raining in Redmond.");
```

For client code, the only added syntax with List<T> compared to ArrayList is the type argument in the declaration and instantiation. In return for this slightly more coding complexity, you can create a list that is not only safer than ArrayList, but also significantly faster, especially when the list items are value types.

See Also

System.Collections.Generic
C# Programming Guide
Introduction to Generics
Boxing and Unboxing
When to Use Generic Collections
Guidelines for Collections

Generic Type Parameters (C# Programming Guide)

5/4/2018 • 2 minutes to read • Edit Online

In a generic type or method definition, a type parameters is a placeholder for a specific type that a client specifies when they instantiate a variable of the generic type. A generic class, such as `GenericList<T>` listed in Introduction to Generics, cannot be used as-is because it is not really a type; it is more like a blueprint for a type. To use `GenericList<T>`, client code must declare and instantiate a constructed type by specifying a type argument inside the angle brackets. The type argument for this particular class can be any type recognized by the compiler. Any number of constructed type instances can be created, each one using a different type argument, as follows:

```
GenericList<float> list1 = new GenericList<float>();
GenericList<ExampleClass> list2 = new GenericList<ExampleClass>();
GenericList<ExampleStruct> list3 = new GenericList<ExampleStruct>();
```

In each of these instances of `GenericList<T>`, every occurrence of `T` in the class will be substituted at run time with the type argument. By means of this substitution, we have created three separate type-safe and efficient objects using a single class definition. For more information on how this substitution is performed by the CLR, see Generics in the Run Time.

Type Parameter Naming Guidelines

- **Do** name generic type parameters with descriptive names, unless a single letter name is completely self explanatory and a descriptive name would not add value.

  ```
  public interface ISessionChannel<TSession> { /*...*/ }
  public delegate TOutput Converter<TInput, TOutput>(TInput from);
  public class List<T> { /*...*/ }
  ```

- **Consider** using T as the type parameter name for types with one single letter type parameter.

  ```
  public int IComparer<T>() { return 0; }
  public delegate bool Predicate<T>(T item);
  public struct Nullable<T> where T : struct { /*...*/ }
  ```

- **Do** prefix descriptive type parameter names with "T".

  ```
  public interface ISessionChannel<TSession>
  {
      TSession Session { get; }
  }
  ```

- **Consider** indicating constraints placed on a type parameter in the name of parameter. For example, a parameter constrained to `ISession` may be called `TSession`.

See Also

System.Collections.Generic
C# Programming Guide
Generics

Constraints on type parameters (C# Programming Guide)

5/22/2018 • 8 minutes to read • Edit Online

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.
`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.

```
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:

```
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.

```
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:

```
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:

```
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:

```
//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:

```
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:

```
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:

```
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.

```
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:

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

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

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

See also

System.Collections.Generic C# Programming Guide

Generic Classes (C# Programming Guide)

5/4/2018 • 3 minutes to read • Edit Online

Generic classes encapsulate operations that are not specific to a particular data type. The most common use for generic classes is with collections like linked lists, hash tables, stacks, queues, trees, and so on. Operations such as adding and removing items from the collection are performed in basically the same way regardless of the type of data being stored.

For most scenarios that require collection classes, the recommended approach is to use the ones provided in the .NET class library. For more information about using these classes, see Generic Collections in .NET.

Typically, you create generic classes by starting with an existing concrete class, and changing types into type parameters one at a time until you reach the optimal balance of generalization and usability. When creating your own generic classes, important considerations include the following:

- Which types to generalize into type parameters.

  As a rule, the more types you can parameterize, the more flexible and reusable your code becomes. However, too much generalization can create code that is difficult for other developers to read or understand.

- What constraints, if any, to apply to the type parameters (See Constraints on Type Parameters).

  A good rule is to apply the maximum constraints possible that will still let you handle the types you must handle. For example, if you know that your generic class is intended for use only with reference types, apply the class constraint. That will prevent unintended use of your class with value types, and will enable you to use the `as` operator on `T`, and check for null values.

- Whether to factor generic behavior into base classes and subclasses.

  Because generic classes can serve as base classes, the same design considerations apply here as with non-generic classes. See the rules about inheriting from generic base classes later in this topic.

- Whether to implement one or more generic interfaces.

  For example, if you are designing a class that will be used to create items in a generics-based collection, you may have to implement an interface such as IComparable<T> where `T` is the type of your class.

For an example of a simple generic class, see Introduction to Generics.

The rules for type parameters and constraints have several implications for generic class behavior, especially regarding inheritance and member accessibility. Before proceeding, you should understand some terms. For a generic class `Node<T>,` client code can reference the class either by specifying a type argument, to create a closed constructed type (`Node<int>`). Alternatively, it can leave the type parameter unspecified, for example when you specify a generic base class, to create an open constructed type (`Node<T>`). Generic classes can inherit from concrete, closed constructed, or open constructed base classes:

```
class BaseNode { }
class BaseNodeGeneric<T> { }

// concrete type
class NodeConcrete<T> : BaseNode { }

//closed constructed type
class NodeClosed<T> : BaseNodeGeneric<int> { }

//open constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }
```

Non-generic, in other words, concrete, classes can inherit from closed constructed base classes, but not from open constructed classes or from type parameters because there is no way at run time for client code to supply the type argument required to instantiate the base class.

```
//No error
class Node1 : BaseNodeGeneric<int> { }

//Generates an error
//class Node2 : BaseNodeGeneric<T> {}

//Generates an error
//class Node3 : T {}
```

Generic classes that inherit from open constructed types must supply type arguments for any base class type parameters that are not shared by the inheriting class, as demonstrated in the following code:

```
class BaseNodeMultiple<T, U> { }

//No error
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> {}
```

Generic classes that inherit from open constructed types must specify constraints that are a superset of, or imply, the constraints on the base type:

```
class NodeItem<T> where T : System.IComparable<T>, new() { }
class SpecialNodeItem<T> : NodeItem<T> where T : System.IComparable<T>, new() { }
```

Generic types can use multiple type parameters and constraints, as follows:

```
class SuperKeyType<K, V, U>
    where U : System.IComparable<U>
    where V : new()
{ }
```

Open constructed and closed constructed types can be used as method parameters:

```
void Swap<T>(List<T> list1, List<T> list2)
{
    //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
    //code to swap items
}
```

If a generic class implements an interface, all instances of that class can be cast to that interface.

Generic classes are invariant. In other words, if an input parameter specifies a `List<BaseClass>`, you will get a compile-time error if you try to provide a `List<DerivedClass>`.

See Also

System.Collections.Generic
C# Programming Guide
Generics
Saving the State of Enumerators
An Inheritance Puzzle, Part One

Generic Interfaces (C# Programming Guide)

5/4/2018 • 4 minutes to read • Edit Online

It is often useful to define interfaces either for generic collection classes, or for the generic classes that represent items in the collection. The preference for generic classes is to use generic interfaces, such as IComparable<T> rather than IComparable, in order to avoid boxing and unboxing operations on value types. The .NET Framework class library defines several generic interfaces for use with the collection classes in the System.Collections.Generic namespace.

When an interface is specified as a constraint on a type parameter, only types that implement the interface can be used. The following code example shows a `SortedList<T>` class that derives from the `GenericList<T>` class. For more information, see Introduction to Generics. `SortedList<T>` adds the constraint `where T : IComparable<T>`. This enables the `BubbleSort` method in `SortedList<T>` to use the generic CompareTo method on list elements. In this example, list elements are a simple class, `Person`, that implements `IComparable<Person>`.

```
//Type parameter T in angle brackets.
public class GenericList<T> : System.Collections.Generic.IEnumerable<T>
{
    protected Node head;
    protected Node current = null;

    // Nested class is also generic on T
    protected class Node
    {
        public Node next;
        private T data;  //T as private member datatype

        public Node(T t)  //T used in non-generic constructor
        {
            next = null;
            data = t;
        }

        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        public T Data  //T as return type of property
        {
            get { return data; }
            set { data = value; }
        }
    }

    public GenericList()  //constructor
    {
        head = null;
    }

    public void AddHead(T t)  //T as method parameter type
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    // Implementation of the iterator
```

```
    public System.Collections.Generic.IEnumerator<T> GetEnumerator()
    {
        Node current = head;
        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    // IEnumerable<T> inherits from IEnumerable, therefore this class
    // must implement both the generic and non-generic versions of
    // GetEnumerator. In most cases, the non-generic method can
    // simply call the generic method.
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

public class SortedList<T> : GenericList<T> where T : System.IComparable<T>
{
    // A simple, unoptimized sort algorithm that
    // orders list elements from lowest to highest:

    public void BubbleSort()
    {
        if (null == head || null == head.Next)
        {
            return;
        }
        bool swapped;

        do
        {
            Node previous = null;
            Node current = head;
            swapped = false;

            while (current.next != null)
            {
                //  Because we need to call this method, the SortedList
                //  class is constrained on IEnumerable<T>
                if (current.Data.CompareTo(current.next.Data) > 0)
                {
                    Node tmp = current.next;
                    current.next = current.next.next;
                    tmp.next = current;

                    if (previous == null)
                    {
                        head = tmp;
                    }
                    else
                    {
                        previous.next = tmp;
                    }
                    previous = tmp;
                    swapped = true;
                }
                else
                {
                    previous = current;
                    current = current.next;
                }
            }
        } while (swapped);
    }
}
```

```
// A simple class that implements IComparable<T> using itself as the
// type argument. This is a common design pattern in objects that
// are stored in generic lists.
public class Person : System.IComparable<Person>
{
    string name;
    int age;

    public Person(string s, int i)
    {
        name = s;
        age = i;
    }

    // This will cause list elements to be sorted on age values.
    public int CompareTo(Person p)
    {
        return age - p.age;
    }

    public override string ToString()
    {
        return name + ":" + age;
    }

    // Must implement Equals.
    public bool Equals(Person p)
    {
        return (this.age == p.age);
    }
}

class Program
{
    static void Main()
    {
        //Declare and instantiate a new generic SortedList class.
        //Person is the type argument.
        SortedList<Person> list = new SortedList<Person>();

        //Create name and age values to initialize Person objects.
        string[] names = new string[]
        {
            "Franscoise",
            "Bill",
            "Li",
            "Sandra",
            "Gunnar",
            "Alok",
            "Hiroyuki",
            "Maria",
            "Alessandro",
            "Raul"
        };

        int[] ages = new int[] { 45, 19, 28, 23, 18, 9, 108, 72, 30, 35 };

        //Populate the list.
        for (int x = 0; x < 10; x++)
        {
            list.AddHead(new Person(names[x], ages[x]));
        }

        //Print out unsorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
```

```
        System.Console.WriteLine("Done with unsorted list");

        //Sort the list.
        list.BubbleSort();

        //Print out sorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with sorted list");
    }
}
```

Multiple interfaces can be specified as constraints on a single type, as follows:

```
class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}
```

An interface can define more than one type parameter, as follows:

```
interface IDictionary<K, V>
{
}
```

The rules of inheritance that apply to classes also apply to interfaces:

```
interface IMonth<T> { }

interface IJanuary     : IMonth<int> { }  //No error
interface IFebruary<T> : IMonth<int> { }  //No error
interface IMarch<T>    : IMonth<T> { }    //No error
//interface IApril<T>  : IMonth<T, U> {}  //Error
```

Generic interfaces can inherit from non-generic interfaces if the generic interface is contra-variant, which means it only uses its type parameter as a return value. In the .NET Framework class library, IEnumerable<T> inherits from IEnumerable because IEnumerable<T> only uses `T` in the return value of GetEnumerator and in the Current property getter.

Concrete classes can implement closed constructed interfaces, as follows:

```
interface IBaseInterface<T> { }

class SampleClass : IBaseInterface<string> { }
```

Generic classes can implement generic interfaces or closed constructed interfaces as long as the class parameter list supplies all arguments required by the interface, as follows:

```
interface IBaseInterface1<T> { }
interface IBaseInterface2<T, U> { }

class SampleClass1<T> : IBaseInterface1<T> { }          //No error
class SampleClass2<T> : IBaseInterface2<T, string> { }  //No error
```

The rules that control method overloading are the same for methods within generic classes, generic structs, or generic interfaces. For more information, see Generic Methods.

See Also

C# Programming Guide
Introduction to Generics
interface
Generics

Generic Methods (C# Programming Guide)

5/4/2018 • 2 minutes to read • Edit Online

A generic method is a method that is declared with type parameters, as follows:

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

The following code example shows one way to call the method by using `int` for the type argument:

```
public static void TestSwap()
{
    int a = 1;
    int b = 2;

    Swap<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
}
```

You can also omit the type argument and the compiler will infer it. The following call to `Swap` is equivalent to the previous call:

```
Swap(ref a, ref b);
```

The same rules for type inference apply to static methods and instance methods. The compiler can infer the type parameters based on the method arguments you pass in; it cannot infer the type parameters only from a constraint or return value. Therefore type inference does not work with methods that have no parameters. Type inference occurs at compile time before the compiler tries to resolve overloaded method signatures. The compiler applies type inference logic to all generic methods that share the same name. In the overload resolution step, the compiler includes only those generic methods on which type inference succeeded.

Within a generic class, non-generic methods can access the class-level type parameters, as follows:

```
class SampleClass<T>
{
    void Swap(ref T lhs, ref T rhs) { }
}
```

If you define a generic method that takes the same type parameters as the containing class, the compiler generates warning CS0693 because within the method scope, the argument supplied for the inner `T` hides the argument supplied for the outer `T`. If you require the flexibility of calling a generic class method with type arguments other than the ones provided when the class was instantiated, consider providing another identifier for the type parameter of the method, as shown in `GenericList2<T>` in the following example.

```
class GenericList<T>
{
    // CS0693
    void SampleMethod<T>() { }
}

class GenericList2<T>
{
    //No warning
    void SampleMethod<U>() { }
}
```

Use constraints to enable more specialized operations on type parameters in methods. This version of `Swap<T>`, now named `SwapIfGreater<T>`, can only be used with type arguments that implement IComparable<T>.

```
void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable<T>
{
    T temp;
    if (lhs.CompareTo(rhs) > 0)
    {
        temp = lhs;
        lhs = rhs;
        rhs = temp;
    }
}
```

Generic methods can be overloaded on several type parameters. For example, the following methods can all be located in the same class:

```
void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }
```

C# Language Specification

For more information, see the C# Language Specification.

See Also

System.Collections.Generic
C# Programming Guide
Introduction to Generics
Methods

Generics and Arrays (C# Programming Guide)

5/4/2018 • 2 minutes to read • Edit Online

In C# 2.0 and later, single-dimensional arrays that have a lower bound of zero automatically implement IList<T>. This enables you to create generic methods that can use the same code to iterate through arrays and other collection types. This technique is primarily useful for reading data in collections. The IList<T> interface cannot be used to add or remove elements from an array. An exception will be thrown if you try to call an IList<T> method such as RemoveAt on an array in this context.

The following code example demonstrates how a single generic method that takes an IList<T> input parameter can iterate through both a list and an array, in this case an array of integers.

```
class Program
{
    static void Main()
    {
        int[] arr = { 0, 1, 2, 3, 4 };
        List<int> list = new List<int>();

        for (int x = 5; x < 10; x++)
        {
            list.Add(x);
        }

        ProcessItems<int>(arr);
        ProcessItems<int>(list);
    }

    static void ProcessItems<T>(IList<T> coll)
    {
        // IsReadOnly returns True for the array and False for the List.
        System.Console.WriteLine
            ("IsReadOnly returns {0} for this collection.",
            coll.IsReadOnly);

        // The following statement causes a run-time exception for the
        // array, but not for the List.
        //coll.RemoveAt(4);

        foreach (T item in coll)
        {
            System.Console.Write(item.ToString() + " ");
        }
        System.Console.WriteLine();
    }
}
```

See Also

System.Collections.Generic
C# Programming Guide
Generics
Arrays
Generics

Generic Delegates (C# Programming Guide)

5/4/2018 • 2 minutes to read • Edit Online

A delegate can define its own type parameters. Code that references the generic delegate can specify the type argument to create a closed constructed type, just like when instantiating a generic class or calling a generic method, as shown in the following example:

```
public delegate void Del<T>(T item);
public static void Notify(int i) { }

Del<int> m1 = new Del<int>(Notify);
```

C# version 2.0 has a new feature called method group conversion, which applies to concrete as well as generic delegate types, and enables you to write the previous line with this simplified syntax:

```
Del<int> m2 = Notify;
```

Delegates defined within a generic class can use the generic class type parameters in the same way that class methods do.

```
class Stack<T>
{
    T[] items;
    int index;

    public delegate void StackDelegate(T[] items);
}
```

Code that references the delegate must specify the type argument of the containing class, as follows:

```
private static void DoWork(float[] items) { }

public static void TestStack()
{
    Stack<float> s = new Stack<float>();
    Stack<float>.StackDelegate d = DoWork;
}
```

Generic delegates are especially useful in defining events based on the typical design pattern because the sender argument can be strongly typed and no longer has to be cast to and from Object.

```
delegate void StackEventHandler<T, U>(T sender, U eventArgs);

class Stack<T>
{
    public class StackEventArgs : System.EventArgs { }
    public event StackEventHandler<Stack<T>, StackEventArgs> stackEvent;

    protected virtual void OnStackChanged(StackEventArgs a)
    {
        stackEvent(this, a);
    }
}

class SampleClass
{
    public void HandleStackChange<T>(Stack<T> stack, Stack<T>.StackEventArgs args) { }
}

public static void Test()
{
    Stack<double> s = new Stack<double>();
    SampleClass o = new SampleClass();
    s.stackEvent += o.HandleStackChange;
}
```

See Also

System.Collections.Generic
C# Programming Guide
Introduction to Generics
Generic Methods
Generic Classes
Generic Interfaces
Delegates
Generics

Differences Between C++ Templates and C# Generics (C# Programming Guide)

5/4/2018 • 2 minutes to read • Edit Online

C# Generics and C++ templates are both language features that provide support for parameterized types. However, there are many differences between the two. At the syntax level, C# generics are a simpler approach to parameterized types without the complexity of C++ templates. In addition, C# does not attempt to provide all of the functionality that C++ templates provide. At the implementation level, the primary difference is that C# generic type substitutions are performed at runtime and generic type information is thereby preserved for instantiated objects. For more information, see Generics in the Run Time.

The following are the key differences between C# Generics and C++ templates:

- C# generics do not provide the same amount of flexibility as C++ templates. For example, it is not possible to call arithmetic operators in a C# generic class, although it is possible to call user defined operators.
- C# does not allow non-type template parameters, such as `template C<int i> {}`.
- C# does not support explicit specialization; that is, a custom implementation of a template for a specific type.
- C# does not support partial specialization: a custom implementation for a subset of the type arguments.
- C# does not allow the type parameter to be used as the base class for the generic type.
- C# does not allow type parameters to have default types.
- In C#, a generic type parameter cannot itself be a generic, although constructed types can be used as generics. C++ does allow template parameters.
- C++ allows code that might not be valid for all type parameters in the template, which is then checked for the specific type used as the type parameter. C# requires code in a class to be written in such a way that it will work with any type that satisfies the constraints. For example, in C++ it is possible to write a function that uses the arithmetic operators `+` and `-` on objects of the type parameter, which will produce an error at the time of instantiation of the template with a type that does not support these operators. C# disallows this; the only language constructs allowed are those that can be deduced from the constraints.

See Also

C# Programming Guide
Introduction to Generics
Templates

Generics in the Run Time (C# Programming Guide)

5/4/2018 • 3 minutes to read • Edit Online

When a generic type or method is compiled into Microsoft intermediate language (MSIL), it contains metadata that identifies it as having type parameters. How the MSIL for a generic type is used differs based on whether the supplied type parameter is a value type or reference type.

When a generic type is first constructed with a value type as a parameter, the runtime creates a specialized generic type with the supplied parameter or parameters substituted in the appropriate locations in the MSIL. Specialized generic types are created one time for each unique value type that is used as a parameter.

For example, suppose your program code declared a stack that is constructed of integers:

```
Stack<int> stack;
```

At this point, the runtime generates a specialized version of the Stack<T> class that has the integer substituted appropriately for its parameter. Now, whenever your program code uses a stack of integers, the runtime reuses the generated specialized Stack<T> class. In the following example, two instances of a stack of integers are created, and they share a single instance of the `Stack<int>` code:

```
Stack<int> stackOne = new Stack<int>();
Stack<int> stackTwo = new Stack<int>();
```

However, suppose that another Stack<T> class with a different value type such as a `long` or a user-defined structure as its parameter is created at another point in your code. As a result, the runtime generates another version of the generic type and substitutes a `long` in the appropriate locations in MSIL. Conversions are no longer necessary because each specialized generic class natively contains the value type.

Generics work somewhat differently for reference types. The first time a generic type is constructed with any reference type, the runtime creates a specialized generic type with object references substituted for the parameters in the MSIL. Then, every time that a constructed type is instantiated with a reference type as its parameter, regardless of what type it is, the runtime reuses the previously created specialized version of the generic type. This is possible because all references are the same size.

For example, suppose you had two reference types, a `Customer` class and an `Order` class, and also suppose that you created a stack of `Customer` types:

```
class Customer { }
class Order { }
```

```
Stack<Customer> customers;
```

At this point, the runtime generates a specialized version of the Stack<T> class that stores object references that will be filled in later instead of storing data. Suppose the next line of code creates a stack of another reference type, which is named `Order`:

```
Stack<Order> orders = new Stack<Order>();
```

Unlike with value types, another specialized version of the Stack<T> class is not created for the `Order` type. Instead, an instance of the specialized version of the Stack<T> class is created and the `orders` variable is set to reference it. Suppose that you then encountered a line of code to create a stack of a `Customer` type:

```
customers = new Stack<Customer>();
```

As with the previous use of the Stack<T> class created by using the `Order` type, another instance of the specialized Stack<T> class is created. The pointers that are contained therein are set to reference an area of memory the size of a `Customer` type. Because the number of reference types can vary wildly from program to program, the C# implementation of generics greatly reduces the amount of code by reducing to one the number of specialized classes created by the compiler for generic classes of reference types.

Moreover, when a generic C# class is instantiated by using a value type or reference type parameter, reflection can query it at runtime and both its actual type and its type parameter can be ascertained.

See Also

System.Collections.Generic
C# Programming Guide
Introduction to Generics
Generics

Generics and Reflection (C# Programming Guide)

5/4/2018 • 2 minutes to read • Edit Online

Because the Common Language Runtime (CLR) has access to generic type information at run time, you can use reflection to obtain information about generic types in the same way as for non-generic types. For more information, see Generics in the Run Time.

In the .NET Framework 2.0 several new members are added to the Type class to enable run-time information for generic types. See the documentation on these classes for more information on how to use these methods and properties. The System.Reflection.Emit namespace also contains new members that support generics. See How to: Define a Generic Type with Reflection Emit.

For a list of the invariant conditions for terms used in generic reflection, see the IsGenericType property remarks.

SYSTEM.TYPE MEMBER NAME	DESCRIPTION
IsGenericType	Returns true if a type is generic.
GetGenericArguments	Returns an array of `Type` objects that represent the type arguments supplied for a constructed type, or the type parameters of a generic type definition.
GetGenericTypeDefinition	Returns the underlying generic type definition for the current constructed type.
GetGenericParameterConstraints	Returns an array of `Type` objects that represent the constraints on the current generic type parameter.
ContainsGenericParameters	Returns true if the type or any of its enclosing types or methods contain type parameters for which specific types have not been supplied.
GenericParameterAttributes	Gets a combination of `GenericParameterAttributes` flags that describe the special constraints of the current generic type parameter.
GenericParameterPosition	For a `Type` object that represents a type parameter, gets the position of the type parameter in the type parameter list of the generic type definition or generic method definition that declared the type parameter.
IsGenericParameter	Gets a value that indicates whether the current `Type` represents a type parameter of a generic type or method definition.
IsGenericTypeDefinition	Gets a value that indicates whether the current Type represents a generic type definition, from which other generic types can be constructed. Returns true if the type represents the definition of a generic type.

SYSTEM.TYPE MEMBER NAME	DESCRIPTION
DeclaringMethod	Returns the generic method that defined the current generic type parameter, or null if the type parameter was not defined by a generic method.
MakeGenericType	Substitutes the elements of an array of types for the type parameters of the current generic type definition, and returns a Type object representing the resulting constructed type.

In addition, members of the MethodInfo class enable run-time information for generic methods. See the IsGenericMethod property remarks for a list of invariant conditions for terms used to reflect on generic methods.

SYSTEM.REFLECTION.MEMBERINFO MEMBER NAME	DESCRIPTION
IsGenericMethod	Returns true if a method is generic.
GetGenericArguments	Returns an array of Type objects that represent the type arguments of a constructed generic method or the type parameters of a generic method definition.
GetGenericMethodDefinition	Returns the underlying generic method definition for the current constructed method.
ContainsGenericParameters	Returns true if the method or any of its enclosing types contain any type parameters for which specific types have not been supplied.
IsGenericMethodDefinition	Returns true if the current MethodInfo represents the definition of a generic method.
MakeGenericMethod	Substitutes the elements of an array of types for the type parameters of the current generic method definition, and returns a MethodInfo object representing the resulting constructed method.

See Also

C# Programming Guide
Generics
Reflection and Generic Types
Generics

Generics and Attributes (C# Programming Guide)

5/4/2018 • 2 minutes to read • Edit Online

Attributes can be applied to generic types in the same way as non-generic types. For more information on applying attributes, see Attributes.

Custom attributes are only permitted to reference open generic types, which are generic types for which no type arguments are supplied, and closed constructed generic types, which supply arguments for all type parameters.

The following examples use this custom attribute:

```
class CustomAttribute : System.Attribute
{
    public System.Object info;
}
```

An attribute can reference an open generic type:

```
public class GenericClass1<T> { }

[CustomAttribute(info = typeof(GenericClass1<>))]
class ClassA { }
```

Specify multiple type parameters using the appropriate number of commas. In this example, `GenericClass2` has two type parameters:

```
public class GenericClass2<T, U> { }

[CustomAttribute(info = typeof(GenericClass2<,>))]
class ClassB { }
```

An attribute can reference a closed constructed generic type:

```
public class GenericClass3<T, U, V> { }

[CustomAttribute(info = typeof(GenericClass3<int, double, string>))]
class ClassC { }
```

An attribute that references a generic type parameter will cause a compile-time error:

```
//[CustomAttribute(info = typeof(GenericClass3<int, T, string>))]  //Error
class ClassD<T> { }
```

A generic type cannot inherit from Attribute:

```
//public class CustomAtt<T> : System.Attribute {}  //Error
```

To obtain information about a generic type or type parameter at run time, you can use the methods of System.Reflection. For more information, see Generics and Reflection

See Also

C# Programming Guide
Generics
Attributes