Generic Interfaces (C# Programming Guide)

Article • 07/09/2022

It's often useful to define interfaces either for generic collection classes, or for the generic classes that represent items in the collection. To avoid boxing and unboxing operations on value types, it's better to use generic interfaces, such as IComparable<T>, on generic classes. The .NET class library defines several generic interfaces for use with the collection classes in the System.Collections.Generic namespace. For more information about these interfaces, see Generic interfaces.

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 constraint 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>`.

C#

```
//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 IComparable<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);
        }
    }

    public class Program
    {
        public 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:

C#

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

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

C#

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

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

C#

```
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 covariant, which means it only uses its type parameter as a return value. In the .NET 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:

C#

```
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:

C#

```
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.

Beginning with C# 11, interfaces may declare `static abstract` or `static virtual` members. Interfaces that declare either `static abstract` or `static virtual` members are almost always generic interfaces. The compiler must resolve calls to `static virtual` and `static abstract` methods at compile time. `static virtual` and `static abstract` methods declared in interfaces don't have a runtime dispatch mechanism analogous to `virtual` or `abstract` methods declared in classes. Instead, the compiler uses type information available at compile time. These members are typically declared in generic interfaces. Furthermore, most interfaces that declare `static virtual` or `static abstract` methods declare that one of the type parameters must implement the declared interface. The compiler then uses the supplied type arguments to resolve the type of the declared member.

See also

- C# Programming Guide
- Introduction to Generics
- interface
- Generics