# Generic classes and methods

Article • 07/11/2023

Generics introduces the concept of type parameters to .NET, 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:

C#

```
// 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());
    }
}
```

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. The System.Collections.Generic namespace contains several generic-based collection classes. The non-generic collections, such as ArrayList are not recommended and are maintained for compatibility purposes. For more information, see Generics in .NET.

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 .NET 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.

`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`.

C#

```
// 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:

C#

```
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");
    }
}
```

> **Note**
>
> Following examples applies not only to `class` types, but also `interface` and `struct` types

## 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 class library contains several generic collection classes in the System.Collections.Generic namespace. The generic collections 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.

## C# language specification

For more information, see the C# Language Specification.

## See also

- System.Collections.Generic
- Generics in .NET