Generics in .NET

Article • 02/17/2023

Generics let you tailor a method, class, structure, or interface to the precise data type it acts upon. For example, instead of using the Hashtable class, which allows keys and values to be of any type, you can use the Dictionary<TKey,TValue> generic class and specify the types allowed for the key and the value. Among the benefits of generics are increased code reusability and type safety.

Define and use generics

Generics are classes, structures, interfaces, and methods that have placeholders (type parameters) for one or more of the types that they store or use. A generic collection class might use a type parameter as a placeholder for the type of objects that it stores. The type parameters appear as the types of its fields and the parameter types of its methods. A generic method might use its type parameter as the type of its return value or as the type of one of its formal parameters.

The following code illustrates a simple generic class definition.

C#

```
public class SimpleGenericClass<T>
{
    public T Field;
}
```

When you create an instance of a generic class, you specify the actual types to substitute for the type parameters. This establishes a new generic class, referred to as a constructed generic class, with your chosen types substituted everywhere that the type parameters appear. The result is a type-safe class that is tailored to your choice of types, as the following code illustrates.

C#

```
public static void Main()
{
    SimpleGenericClass<string> g = new SimpleGenericClass<string>();
    g.Field = "A string";
    //...
    Console.WriteLine("SimpleGenericClass.Field           = \"{0}\"",
g.Field);
    Console.WriteLine("SimpleGenericClass.Field.GetType() = {0}", g.-
Field.GetType().FullName);
}
```

Terminology

The following terms are used to discuss generics in .NET:

- A *generic type definition* is a class, structure, or interface declaration that functions as a template, with placeholders for the types that it can contain or use. For example, the System.Collections.Generic.Dictionary<TKey,TValue> class can contain two types: keys and values. Because a generic type definition is only a template, you cannot create instances of a class, structure, or interface that is a generic type definition.

- *Generic type parameters*, or *type parameters*, are the placeholders in a generic type or method definition. The System.Collections.Generic.Dictionary<TKey,TValue> generic type has two type parameters, `TKey` and `TValue`, that represent the types of its keys and values.

- A *constructed generic type*, or *constructed type*, is the result of specifying types for the generic type parameters of a generic type definition.

- A *generic type argument* is any type that is substituted for a generic type parameter.

- The general term *generic type* includes both constructed types and generic type definitions.

- *Covariance* and *contravariance* of generic type parameters enable you to use constructed generic types whose type arguments are more derived (covariance) or less derived (contravariance) than a target constructed type. Covariance and contravariance are collectively referred to as *variance*. For more information, see Covariance and contravariance.

- *Constraints* are limits placed on generic type parameters. For example, you might limit a type parameter to types that implement the System.Collections.Generic.IComparer<T> generic interface, to ensure that instances of the type can be ordered. You can also constrain type parameters to types that have a particular base class, that have a parameterless constructor, or that are reference types or value types. Users of the generic type cannot substitute type arguments that do not satisfy the constraints.

- A *generic method definition* is a method with two parameter lists: a list of generic type parameters and a list of formal parameters. Type parameters can appear as the return type or as the types of the formal parameters, as the following code shows.

  C#

  ```
  T MyGenericMethod<T>(T arg)
  {
      T temp = arg;
      //...
      return temp;
  }
  ```

  Generic methods can appear on generic or nongeneric types. It's important to note that a method is not generic just because it belongs to a generic type, or even because it has formal parameters whose types are the generic parameters of the enclosing type. A method is generic only if it has its own list of type parameters. In the following code, only method `G` is generic.

  C#

  ```
  class A
  {
      T G<T>(T arg)
      {
          T temp = arg;
          //...
          return temp;
      }
  }
  class MyGenericClass<T>
  {
      T M(T arg)
      {
          T temp = arg;
          //...
          return temp;
      }
  }
  ```

Advantages and disadvantages of generics

There are many advantages to using generic collections and delegates:

- Type safety. Generics shift the burden of type safety from you to the compiler. There is no need to write code to test for the correct data type because it is enforced at compile time. The need for type casting and the possibility of run-time errors are reduced.

- Less code and code is more easily reused. There is no need to inherit from a base type and override members. For example, the LinkedList<T> is ready for immediate use. For example, you can create a linked list of strings with the following variable declaration:

  C#

  ```
  LinkedList<string> llist = new LinkedList<string>();
  ```

- Better performance. Generic collection types generally perform better for storing and manipulating value types because there is no need to box the value types.

- Generic delegates enable type-safe callbacks without the need to create multiple delegate classes. For example, the Predicate<T> generic delegate allows you to create a method that implements your own search criteria for a particular type and to use your method with methods of the Array type such as Find, FindLast, and FindAll.

- Generics streamline dynamically generated code. When you use generics with dynamically generated code you do not need to generate the type. This increases the number of scenarios in which you can use lightweight dynamic methods instead of generating entire assemblies. For more information, see How to: Define and Execute Dynamic Methods and DynamicMethod.

The following are some limitations of generics:

- Generic types can be derived from most base classes, such as MarshalByRefObject (and constraints can be used to require that generic type parameters derive from base classes like MarshalByRefObject). However, .NET does not support context-bound generic types. A generic type can be derived from ContextBoundObject, but trying to create an instance of that type causes a TypeLoadException.

- Enumerations cannot have generic type parameters. An enumeration can be generic only incidentally (for example, because it is nested in a generic type that is defined using Visual Basic, C#, or C++). For more information, see "Enumerations" in Common Type System.

- Lightweight dynamic methods cannot be generic.

- In Visual Basic, C#, and C++, a nested type that is enclosed in a generic type cannot be instantiated unless types have been assigned to the type parameters of all enclosing types. Another way of saying this is that in reflection, a nested type that is defined using these languages includes the type parameters of all its enclosing types. This allows the type parameters of enclosing types to be used in the member definitions of a nested type. For more information, see "Nested Types" in MakeGenericType.

ⓘ **Note**

A nested type that is defined by emitting code in a dynamic assembly or by using the **Ilasm.exe (IL Assembler)** is not required to include the type parameters of its enclosing types; however, if it does not include them, the type parameters are not in scope in the nested class.

For more information, see "Nested Types" in MakeGenericType.

Class library and language support

.NET provides a number of generic collection classes in the following namespaces:

- The System.Collections.Generic namespace contains most of the generic collection types provided by .NET, such as the List<T> and Dictionary<TKey,TValue> generic classes.
- The System.Collections.ObjectModel namespace contains additional generic collection types, such as the ReadOnlyCollection<T> generic class, that are useful for exposing object models to users of your classes.

Generic interfaces for implementing sort and equality comparisons are provided in the System namespace, along with generic delegate types for event handlers, conversions, and search predicates.

The System.Numerics namespace provides generic interfaces for mathematical functionality (available in .NET 7 and later versions). For more information, see Generic math.

Support for generics has been added to the System.Reflection namespace for examining generic types and generic methods, to System.Reflection.Emit for emitting dynamic assemblies that contain generic types and methods, and to System.CodeDom for generating source graphs that include generics.

The common language runtime provides new opcodes and prefixes to support generic types in Microsoft intermediate language (MSIL), including Stelem, Ldelem, Unbox_Any, Constrained, and Readonly.

Visual C++, C#, and Visual Basic all provide full support for defining and using generics. For more information about language support, see Generic Types in Visual Basic, Introduction to Generics, and Overview of Generics in Visual C++.

Nested types and generics

A type that is nested in a generic type can depend on the type parameters of the enclosing generic type. The common language runtime considers nested types to be generic, even if they do not have generic type parameters of their own. When you create an instance of a nested type, you must specify type arguments for all enclosing generic types.

Related articles

Title	Description
Generic Collections in .NET	Describes generic collection classes and other generic types in .NET.
Generic Delegates for Manipulating Arrays and Lists	Describes generic delegates for conversions, search predicates, and actions to be taken on elements of an array or collection.
Generic math	Describes how you can perform mathematical operations generically.
Generic Interfaces	Describes generic interfaces that provide common functionality across families of generic types.
Covariance and Contravariance	Describes covariance and contravariance in generic type parameters.
Commonly Used Collection Types	Provides summary information about the characteristics and usage scenarios of the collection types in .NET, including generic types.
When to Use Generic Collections	Describes general rules for determining when to use generic collection types.
How to: Define a Generic Type with Reflection Emit	Explains how to generate dynamic assemblies that include generic types and methods.
Generic Types in Visual Basic	Describes the generics feature for Visual Basic users, including how-to topics for using and defining generic types.

Title	Description
Introduction to Generics	Provides an overview of defining and using generic types for C# users.
Overview of Generics in Visual C++	Describes the generics feature for C++ users, including the differences between generics and templates.

Reference

- System.Collections.Generic
- System.Collections.ObjectModel
- System.Reflection.Emit.OpCodes