Answer on Question#39017- Programming, C#

1. which features of c# net allow reuse the existing application code?

Solution.

Generics in the .NET Framework

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 type allowed for the key and the type allowed for the value. Among the benefits of generics are increased code reusability and type safety.

Defining and Using 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 Generic<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()
{
    Generic<string> g = new Generic<string>();
    g.Field = "A string";
    //...
    Console.WriteLine("Generic.Field           = \"{0}\"", g.Field);
    Console.WriteLine("Generic.Field.GetType() = {0}",
g.Field.GetType().FullName);
}
```

Back to top

Generics Terminology

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

- 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, theSystem.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. TheSystem.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 in Generics.
- *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 default 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.

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

Generic methods can appear on generic or nongeneric types. It is 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.

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

Class Library and Language Support

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

- The System.Collections.Generic namespace catalogs most of the generic collection types provided by the .NET Framework, such as the List<T> and Dictionary<TKey, TValue> generic classes.
- The System.Collections.ObjectModel namespace catalogs additional generic collection types, such as theReadOnlyCollection<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.
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 toSystem.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 (Visual Basic), Introduction to Generics (C# Programming Guide), 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.