عنوان مضمون

Visual Programming-II

توسط : صفری

بهار1397

generics

Motivation for Generic Methods

```
static void Main( string[] args )
{
        // create arrays of int, double and char
         int[] intArray = { 1, 2, 3, 4, 5, 6 };
         double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
        char[] charArray = { 'H', 'E', 'L', 'L', 'O' };
        Console.WriteLine( "Array intArray contains:" );
        DisplayArray( intArray );
        Console.WriteLine( "Array doubleArray contains:" );
        DisplayArray( doubleArray );
        Console.WriteLine( "Array charArray contains:" );
        DisplayArray( charArray );
} // end Main
```

Motivation for Generic Methods

```
private static void DisplayArray( int[] inputArray )
{       foreach ( int element in inputArray )
        Console.Write( element + " " );
        Console.WriteLine( "\n" );
} // end method DisplayArray
private static void DisplayArray( double[] inputArray )
{       foreach ( double element in inputArray )
        Console.Write( element + " " );
        Console.WriteLine( "\n" );
} // end method DisplayArray
private static void DisplayArray( char[] inputArray )
{       foreach ( char element in inputArray )
        Console.Write( element + " " );
        Console.WriteLine( "\n" );
} // end method DisplayArray
```

Motivation for Generic Methods

When the compiler encounters a method call, it attempts to locate a method declaration that has the ***same* method name** and parameters that *match* the **argument types** in the method call.

Note that the array element type (int, double or char) appears in two locations in each method

1. the method header
2. the foreach statement header.

Motivation for Generic Methods(this code will *not* compile.)

- If we replace the element types in each method with a generic name (such as T for “type”) then all three methods would look like this one

```
private static void DisplayArray( T[] inputArray )
{
        foreach ( T element in inputArray )
        Console.Write( element + " " );
        Console.WriteLine( "\n" );
} // end method DisplayArray
```

Generic-Method Implementation

- You can write a single generic-method declaration that can be called at different times with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.

```
static void Main( string[] args )
{
        // create arrays of int, double and char
         int[] intArray = { 1, 2, 3, 4, 5, 6 };
         double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
        char[] charArray = { 'H', 'E', 'L', 'L', 'O' };
        Console.WriteLine( "Array intArray contains:" );
        DisplayArray( intArray );
        Console.WriteLine( "Array doubleArray contains:" );
        DisplayArray( doubleArray );
        Console.WriteLine( "Array charArray contains:" );
        DisplayArray( charArray );
} // end Main
```

Generic-Method Implementation

```
private static void DisplayArray< T >( T[] inputArray )
{
        foreach ( T element in inputArray )
        Console.Write( element + " " );
        Console.WriteLine( "\n" );
} // end method DisplayArray
```

- All generic method declarations have a **type-parameter list** delimited by angle brackets (<T> in this example) that follows the method's name.
- A type parameter is an identifier that's used in place of actual type names. The type parameters can be used to declare the return type, the parameter types and the local variable types in a generic method declaration; the type parameters act as placeholders for **type arguments** that represent the types of data that will be passed to the generic method.
- You can also use **explicit type arguments** to indicate the exact type that should be used to call a generic function. For example,
- DisplayArray< **int** >(intArray); // pass an int array argument

Generic Classes

- A generic class provides a means for describing a class in a type independent manner. We can then instantiate type-specific versions of the generic class. This capability is an opportunity for software reusability.
- With a generic class, you can use a simple, concise notation to indicate the actual type(s) that should be used in place of the class's type parameter(s). At compilation time, the compiler ensures your code's type safety, and the runtime system replaces type parameters with type arguments to enable your client code to interact with the generic class.

Generic Classes

- One generic Stack class, for example, could be the basis for creating many Stack classes (e.g., “Stack of double,” “Stack of int,” “Stack of char,” “Stack of Employee”). Next slide presents a generic Stack class declaration.
- This class should not be confused with the class Stack from namespace System.Collections.Generics.
- A generic class declaration is similar to a nongeneric class declaration, except that the class name is followed by a type-parameter list and, optionally, one or more constraints on its type parameter.

```
class Stack< T >
{       private int top; // location of the top element
        private T[] elements; // array that stores stack elements
         // parameterless constructor creates a stack of the default size
        public Stack(): this( 10 ) // default stack size
        {// empty constructor; calls constructor at line 18 to perform init
        } // end stack constructor
        // constructor creates a stack of the specified number of elements
        public Stack( int stackSize )
        {       if ( stackSize > 0 ) // validate stackSize
                elements = new T[ stackSize ]; // create stackSize elements
                else
                 throw new ArgumentException( "Stack size must be positive." );
                top = -1; // stack initially empty
       } // end stack constructor
```

```
        public void Push( T pushValue )
        {       if ( top == elements.Length - 1 ) // stack is full
                throw new FullStackException( string.Format("Stack is full, cannot push
{0}", pushValue ) );
               ++top; // increment top
                elements[ top ] = pushValue; // place pushValue on stack
         } // end method Push
        // return the top element if not empty,
        // else throw EmptyStackException
        public T Pop
        {       if ( top == -1 ) // stack is empty
                throw new EmptyStackException( "Stack is empty, cannot pop" );
                 --top; // decrement top
                return elements[ top + 1 ]; // return top value
        } // end method Pop
} // end class Stack
```

Type Constraints

we present a generic Maximum method that determines and returns the largest of its three arguments (all of the same type).

The generic method in this example uses the type parameter to declare *both* the method's **return type** *and* **its parameters**.

Normally, when comparing values to determine which one is greater, you would use the > operator.

However, this operator is not overloaded for use with every type that's built into the Framework Class Library or that might be defined by extending those types.

Type Constraints

Generic code is restricted to performing operations that are guaranteed to work for every possible type.

- we can restrict the types that can be used with a generic method or class to ensure that they meet certain requirements. This feature—known as a **type constraint**—restricts the type of the argument supplied to a particular type parameter.
- ***IComparable<T> Interface***
- It’s possible to compare two objects of the *same* type if that type implements the generic interface **IComparable<T>** (of namespace System).

Type Constraints

```
public static void Main( string[] args )
{       Console.WriteLine( "Maximum of {0}, {1} and {2} is {3}\n",
        3, 4, 5, Maximum( 3, 4, 5 ) );
        Console.WriteLine( "Maximum of {0}, {1} and {2} is {3}\n",
        6.6, 8.8, 7.7, Maximum( 6.6, 8.8, 7.7 ) );
        Console.WriteLine( "Maximum of {0}, {1} and {2} is {3}\n",
         "pear", "apple", "orange",
        Maximum( "pear", "apple", "orange" ) );
} // end Main
```

Type Constraints

```
private static T Maximum< T >( T x, T y, T z ) where T : IComparable< T >
 {       T max = x; // assume x is initially the largest
         // compare y with max
         if ( y.CompareTo( max ) > 0 )
         max = y; // y is the largest so far
         // compare z with max
         if ( z.CompareTo( max ) > 0 )
         max = z; // z is the largest
         return max; // return largest object
} // end method Maximum
```

Type Constraints

C# provides several kinds of type constraints.

- A **class constraint** indicates that the type argument must be an object of a specific base class or one of its subclasses.

```
class B {}
class E<T> where T : B {} // be/derive from base class
```

- An **interface constraint** indicates that the type argument's class must implement a specific interface.

```
interface I {}
class G<T> where T : I {} // be/implement interface
```

Type Constraints

- you can specify a **constructor constraint**—**new()**—to indicate that the generic code can use operator new to create new objects of the type represented by the type parameter.

```
class H<T> where T : new() {} // public default constructor
```

- Finally, You can specify that the type argument must be a reference type or a value type by using the **reference-type constraint** (**class**) or the **value-type constraint** (**struct**), respectively.

```
class C<T> where T : struct {} // value type
class D<T> where T : class {} // reference type
```

Type Constraints

- It’s possible to apply **multiple constraints** to a type parameter. To do so, simply provide a comma-separated list of constraints in the where clause. If you have a class constraint, reference-type constraint or value-type constraint, it must be listed first—only one of these types of constraints can be used for each type parameter. Interface constraints (if any) are listed next. The constructor constraint is listed last (if there is one).

```
class J<T, U>
  where T : class, I
  where U : I, new() {}
```

- for example applying a base class constraint the accessible members of that base class also become available.

```
class Person
{
        public string name;
}
class PersonNameBox<T> where T : Person
{
        public string box;
         public void StorePersonName(T a)
         {
           box = a.name;
         }
}
```

Overloading Generic Methods

- Each overloaded method must have a unique signature.
- A generic method may be **overloaded**.
- A class can provide two or more generic methods with the *same* name but *different* method parameters.
- For example, we could provide a second version of generic method DisplayArray with the additional parameters lowIndex and highIndex that specify the portion of the array to output.

Overloading Generic Methods

- A generic method can be overloaded by nongeneric methods with the same method name.
- When the compiler encounters a method call, it searches for the method declaration that best matches the method name and the argument types specified in the call. For example, generic method DisplayArray of could be overloaded with a version specific to strings that outputs the strings in tabular format . If the compiler cannot match a method call to either a nongeneric method or a generic method, or if there’s ambiguity due to multiple possible matches, the compiler generates an error.