عنوان مضمون

Visual Programming-I

توسط : صفری

خزان1397

methods and Arrays

Functions

- Functions in C# are a means of providing blocks of code that can be executed at any point in an application.

advantages:

1.They let you divide complicated programs into **manageable** pieces.

2.While working on one function, you can **focus on just that part** of the program and construct it, debug it, and perfect it.

3.Different people can **work on different functions simultaneously.**

4.you can write it once and **use it many times**.

5.Using functions greatly enhances the program's **readability** because it reduces the complexity of the function main.

DEFINING AND USING FUNCTIONS

- In code terms, this looks like the following in a console application function of the type you've been looking at:

static <*returnType*> <*FunctionName*>(<*paramType*> <*paramName*>, ...)

{ ...

return <*returnValue*>;

}

- The function definition consists of the following:

➤ Two keywords: static and void

➤ A function name followed by parentheses

➤ A block of code to execute, enclosed in curly braces

DEFINING AND USING FUNCTIONS

Add the following code to Program.cs:

```
class Program
{       static void Write()
        {       WriteLine("Text output from function.");
        }
        static void Main(string[] args)
        {       Write();
                ReadKey();
        }
}
```

Return Values

- When a function returns a value, you have to modify your function in two ways:

➤ Specify the type of the return value in the function declaration instead of using the void keyword.

➤ Use the return keyword to end the function execution and transfer the return value to the calling code.

- This might be as simple as the following:

```
static double GetVal()
{       return 3.2;
}
```

Return Values

- Functions that execute a **single line** of code can use a feature introduced in C# 6 called **expression bodied methods**. The following function pattern uses a => (lambda arrow) to implement this feature.

  static <*returnType*> <*FunctionName*>() => < single line of code >;

- For example, a Multiply() function which prior to C# 6 is written like this:

```
static double Multiply(double myVal1, double myVal2)
{       return myVal1 * myVal2;
}
```

- Can now be written using the => (lambda arrow). The result of the code written here expresses the intent of the method in a much simpler and consolidated way.

```
static double Multiply(double myVal1, double myVal2) => mVal1 * MyVal2;
```

Parameters

- When a function needs to accept parameters, you must specify the following:

➤ A list of the parameters accepted by the function in its definition, along with the types of those parameters

➤ A matching list of arguments in each function call

Try It Out

```
class Program
{
        static int MaxValue(int[] intArray)
        {       int maxVal = intArray[0];
                for (int i = 1; i < intArray.Length; i++)
                {       if (intArray[i] > maxVal)
                        maxVal = intArray[i];
                }
                return maxVal;
        }
        static void Main(string[] args)
        {       int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
                int maxVal = MaxValue(myArray);
                WriteLine($"The maximum value in myArray is {maxVal}");
                ReadKey();
        }
}
```

Parameter Arrays

- When you call a function, you must supply arguments that match the parameters as specified in the function definition. This means matching
  - the parameter types,
  - the number of parameters,
  - the order of the parameters.
- For example, the function

```
static void MyFunction(string myString, double myDouble)
{       ...
}
```

- can’t be called using the following:

```
MyFunction(2.6, "Hello");
```

paramarrays

- At times, while declaring a method, **you are not sure of the number of arguments passed as a parameter**. C# paramarrays (or parameter arrays) come into help at such times
- This parameter, which must be **the last parameter** in the function definition, is known as a *parameter array*.
- Parameter arrays can be a useful way to simplify your code because you don’t have to pass arrays from your calling code. Instead, you pass several arguments of the same type, which are placed in an array you can use from within your function.

paramarrays

- The following code is required to define a function that uses a parameter array:

static <*returnType*> <*FunctionName*>(<*p1Type*> <*p1Name*>, ..., params <*type*>[] <*name*>)

{

...

return <*returnValue*>;

}

You can call this function using code like the following:

<*FunctionName*>(<*p1*>, ..., <*val1*>, <*val2*>, ...)

- <*val1*>, <*val2*>, and so on are values of type <*type*>, which are used to initialize the <*name*> array.

paramarrays

```
class Program
{       static int SumVals(params int[] vals)
        {       int sum = 0;
                foreach (int val in vals)
                {       sum += val;
                }
                return sum;
        }
        static void Main(string[] args)
        {       int sum = SumVals(1, 5, 2, 9, 8);
                WriteLine($"Summed Values = {sum}");
                ReadKey();
        }
}
```

Reference and Value Parameters

- Value Parameters:

  That is, when you have used parameters, you have passed a value into a variable used by the function. Any changes made to this variable in the function have no effect on the argument specified in the function call.

- Reference Parameters:

  the function will work with exactly the same variable as the one used in the function call, not just a variable that has the same value. Any changes made to this variable will, therefore, be reflected in the value of the variable used as an argument.

Value Parameters:

```
static void ShowDouble(int val)
{       val *= 2;
        WriteLine($"val doubled = {val}");
}
```

- Here, the parameter, val, is doubled in this function. If you call it like this,

```
int myNumber = 5;
WriteLine($"myNumber = {myNumber}");
ShowDouble(myNumber);
WriteLine($"myNumber = {myNumber}");
```

Reference Parameters:

```
static void ShowDouble(ref int val)
{       val *= 2;
        WriteLine($"val doubled = {val}");
}
```

- Then, specify it again in the function call (this is mandatory):

```
int myNumber = 5;
WriteLine($"myNumber = {myNumber}");
ShowDouble(ref myNumber);
WriteLine($"myNumber = {myNumber}");
```

Reference and Value Parameters

- **Note** two limitations on the variable used as a ref parameter:
- First, the function might result in a change to the value of a reference parameter, so you must use **a *nonconstant* variable** in the function call. The following is therefore illegal:

```
const int myNumber = 5;
WriteLine($"myNumber = {myNumber}");
ShowDouble(ref myNumber);
WriteLine($"myNumber = {myNumber}");
```

Reference and Value Parameters

- **Note** two limitations on the variable used as a ref parameter:
- Second, you must use **an initialized variable**. C# doesn't allow you to assume that a ref parameter will be initialized in the function that uses it. The following code is also illegal:

```
int myNumber;
ShowDouble(ref myNumber);
WriteLine("myNumber = {myNumber}");
```

Reference and Value Parameters

- It is also possible to use **the ref keyword as a return type**.
- Notice in the following code the ref keyword identifies the return type as ref int, and is also in the code body, which instructs the function to return ref val.

```
static ref int ShowDouble(int val)
{       val *= 2;
        return ref val;
}
```

If you attempted to compile the previous function you would **receive an error.**

Reference and Value Parameters

- The reason is that you cannot pass a variable type as a function parameter by reference without prefixing the ref keyword to the variable declaration. See the following code snippet where the ref keyword is added—that function would compile and run as expected.

```
static ref int ShowDouble(ref int val)
{       val *= 2;
        return ref val;
}
```

Out Parameters

- you can specify that a given parameter is an ***out*** *parameter* by using the **out** keyword, which is used in the **same way as the ref keyword** (as a modifier to the parameter in the function definition and in the function call).
- However, there are important differences:

➤ Whereas it is illegal to use an unassigned variable as a ref parameter, you can use an unassigned variable as an out parameter.

➤ An out parameter must be treated as an unassigned value by the function that uses it.

This means that while it is permissible in calling code to use an assigned variable as an out parameter, the value stored in this variable is lost when the function executes.

Out Parameters

```
static int MaxValue(int[] intArray, out int maxIndex)
{       int maxVal = intArray[0];
        maxIndex = 0;
        for (int i = 1; i < intArray.Length; i++)
        {       if (intArray[i] > maxVal)
                {       maxVal = intArray[i];
                        maxIndex = i;
                }
        }
        return maxVal;
}
```

- You might use the function like this:

```
        int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
        WriteLine("The maximum value in myArray is " +$"{MaxValue(myArray, out int maxIndex)}");
        WriteLine("The first occurrence of this value is " +$" at element {maxIndex + 1}");
```

VARIABLE SCOPE

- The variables in C# are accessible only from localized regions of code. A given variable is said to have a ***scope*** from which it is accessible.

```
class Program
{           static void Write()
            {           WriteLine($"myString = {myString}");
            }
            static void Main(string[] args)
            {           string myString = "String defined in Main()";
                        Write();
                        ReadKey();
            }
}
```

- Compile the code and note the error and warning that appear in the error list:

The name 'myString' does not exist in the current context The variable 'myString' is assigned but its value is never used

VARIABLE SCOPE

- Variables whose scopes cover a single function in this way are known as ***local variables***.
- It is also possible to have ***global variables***, whose scopes cover multiple functions. Modify the code as follows:

```
class Program
{           static string myString;
            static void Write()
            {           string myString = "String defined in Write()";
                        WriteLine("Now in Write()");
                        WriteLine($"Local myString = {myString}");
                        WriteLine($"Global myString = {Program.myString}");
            }
            static void Main(string[] args)
            {           string myString = "String defined in Main()";
                        Program.myString = "Global string";
                        Write();
                        WriteLine("\nNow in Main()");
                        WriteLine($"Local myString = {myString}");
                        WriteLine($"Global myString = {Program.myString}");
                        ReadKey();
            }
}
```

VARIABLE SCOPE

- You might be wondering why you shouldn’t just use this technique to exchange data with functions, rather than the parameter passing shown earlier.

- For example, a global variable can be written to and read from numerous methods within a class or from different threads. Can you be certain that the value in the global variable contains valid data if numerous threads and methods can write to it?

- Without some extra synchronization code, **the answer is probably not**. Additionally, over time it is possible the actual intent of the global variable is forgotten and used later for some other reason. Therefore, the choice of whether to use global variables depends on the intended use of the function in question.

Variable Scope in Other Structures

- Consider the following code:

```
int i;
for (i = 0; i < 10; i++)
{       string text = $"Line {Convert.ToString(i)}";
        WriteLine($"{text}");
}
WriteLine($"Last text output in loop: {text}");
```

- Here, the string variable text is local to the for loop**. This code won't compile** because the call to WriteLine() that occurs outside of this loop attempts to use the variable text, which is out of scope outside of the loop.

Variable Scope in Other Structures

- you can make the following change:

```
int i;
string text = "";
for (i = 0; i < 10; i++)
{       text = $"Line {Convert.ToString(i)}";
        WriteLine($"{text}");
}
WriteLine($"Last text output in loop: {text}");
```

THE MAIN() FUNCTION

- you saw that Main() is the entry point for a C# application and that execution of this function encompasses the execution of the application. That is, when execution is initiated, the Main() function executes, and when the Main() function finishes, execution ends.
- The Main() function can return either void or int, and can optionally include a string[] args parameter, so you can use any of the following versions:

```
static void Main()
static void Main(string[] args)
static int Main()
static int Main(string[] args)
```

THE MAIN() FUNCTION

- The third and fourth versions **return an int value**, which can be used to signify how the application terminates, and often is used as an indication of an error (although this is by no means mandatory).
- In general, **returning a value of 0 reflects normal termination** (that is, the application has completed and can terminate safely).

THE MAIN() FUNCTION

- The optional args parameter of Main() provides you with a way to obtain information from outside the application, specified at runtime. This information takes the form of ***command-line parameters***.

- When a console application is executed, any specified command-line parameters are placed in this args array. You can then use these parameters in your application.

THE MAIN() FUNCTION

```
class Program
{       static void Main(string[] args)
        {       WriteLine($"{args.Length} command line arguments were specified:");
                foreach (string arg in args)
                        WriteLine(arg);
                ReadKey();
        }
}
```

- Open the property pages for the project (right-click on the project name in the Solution Explorer window and select Properties).
- Select the Debug page and add any command-line arguments you want to the Command Line Arguments setting.

STRUCT FUNCTIONS

- The struct types for storing multiple data elements in one place. Structs are actually capable of a lot more than this. For example, they can contain functions as well as data.
- That might seem a little strange at first, but it is, in fact, very useful. As a simple example, consider the following struct:

```
struct CustomerName
{       public string firstName, lastName;
}
```

STRUCT FUNCTIONS

- If you have variables of type CustomerName and you want to output a full name to the console

```
CustomerName myCustomer;
myCustomer.firstName = "John";
myCustomer.lastName = "Franklin";
WriteLine($"{myCustomer.firstName} {myCustomer.lastName}");
```

STRUCT FUNCTIONS

- By adding functions to structs, you can simplify this by centralizing the processing of common tasks. For example, you can add a suitable function to the struct type as follows:

```
struct CustomerName
{       public string firstName, lastName;
public string Name() => firstName + " " + lastName;
}
```

- You can use this function as follows:

```
CustomerName myCustomer;
myCustomer.firstName = "John";
myCustomer.lastName = "Franklin";
WriteLine(myCustomer.Name());
```

USING DELEGATES

- C# delegates are similar to pointers to functions, in C or C++. A **delegate** is a reference type variable that holds the reference to a method.
- Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the **System.Delegate** class.
- The most important purpose of delegates will become clear later in the book when you look at **events and event handling**, but it's useful to briefly consider them here.
- Delegates are declared much like functions, but with no function body and using the delegate keyword. The delegate declaration specifies a return type and parameter list.

  {access-modifier} delegate {return-type} {name}([parameters]);

USING DELEGATES

For example, consider a delegate :

```
public delegate int MethodPointer(int number1, int number2);
```

The preceding delegate can be used to reference any method that have two int parameter and returns an int type variable.

- After defining a delegate, you can **declare a variable** with the type of that delegate. You can then initialize the variable as a reference to any function that has the same return type and parameter list as that delegate. Once you have done this, you can call that function by using the delegate variable as if it were a function.

USING DELEGATES

- Once a delegate type is declared, a delegate object must be created with the **new** keyword and be associated with a particular method. When creating a delegate, the argument passed to the **new** expression is written similar to a method call, but without the arguments to the method. For example :

```
public delegate int MethodPointer(int number1, int number2);
public static int Sum(int number1, int number2)
{
   return number1 + number2;
}
static void Main(string[] args)
{
   MethodPointer pointer = new MethodPointer( Sum);
   Console.WriteLine(pointer(2, 5));
   Console.ReadKey();
}
```

```
class Program
{       delegate double ProcessDelegate(double param1, double param2);
        static double Multiply(double param1, double param2) => param1 * param2;
        static double Divide(double param1, double param2) => param1 / param2;
        static void Main(string[] args)
        {       ProcessDelegate process;
                WriteLine("Enter 2 numbers separated with a comma:");
                string input = ReadLine();
                int commaPos = input.IndexOf(',');
                double param1 = ToDouble(input.Substring(0, commaPos));
                double param2 = ToDouble(input.Substring(commaPos + 1,input.Length - commaPos - 1));
                WriteLine("Enter M to multiply or D to divide:");
                input = ReadLine();
                if (input == "M")
                        process = new ProcessDelegate(Multiply);
                else
                        process = new ProcessDelegate(Divide);
                WriteLine($"Result: {process(param1, param2)}");
                ReadKey();
        }
}
```

Multicasting of a Delegate

- Delegate objects can be composed using the "+" operator. A composed delegate calls the two delegates it was composed from. Only delegates of the same type can be composed. The "-" operator can be used to remove a component delegate from a composed delegate.
- Using this property of delegates you can create an invocation list of methods that will be called when a delegate is invoked. This is called **multicasting** of a delegate. The next slide program demonstrates multicasting of a delegate

Multicasting of a Delegate

```
public delegate void MethodPointer(int number1, int number2);
public static void Sum(int number1, int number2)
   {      Console.WriteLine(number1 + number2);
   }
public static void Substract(int number1, int number2)
   {       Console.WriteLine(number1 - number2);
   }
static void Main(string[] args)
   {      MethodPointer pointer = new MethodPointer( Sum);
          pointer += Substract;
          pointer(8, 3);
          Console.ReadKey();
   }
```