Core C# Programming: Methods, Arrays, Types, Structures

CSIS 3540

Client Server Systems

Class 02

Topics

- Methods
- Arrays
- Enum
- Structure
- Value and Reference Types
- Nullable Types

Methods

- Form:
  - Modifier return_type method_name(params) { body }
  - Where Modifier is one of static, public, virtual, private, …
- Use of **static** keyword allows the method to be called directly without use of a constructor
  - Used where Main() is located.
  - static int Add(int x, int y) { return x + y;}
- Classes only consisting of static methods are commonly know as utility classes.

Parameters

Table 4-1. *C# Parameter Modifiers*

Parameter Modifier	Meaning in Life
(None)	If a parameter is not marked with a parameter modifier, it is assumed to be passed by value, meaning the called method receives a copy of the original data.
`out`	Output parameters must be assigned by the method being called and, therefore, are passed by reference. If the called method fails to assign output parameters, you are issued a compiler error.
`ref`	The value is initially assigned by the caller and may be optionally modified by the called method (as the data is also passed by reference). No compiler error is generated if the called method fails to assign a `ref` parameter.
`params`	This parameter modifier allows you to send in a variable number of arguments as a single logical parameter. A method can have only a single `params` modifier, and it must be the final parameter of the method. In reality, you might not need to use the `params` modifier all too often; however, be aware that numerous methods within the base class libraries do make use of this C# language feature.

To illustrate the use of these keywords, create a new Console Application project named FunWithMethods. Now, let's walk through the role of each keyword.

Parameters

- out vs ref
  - Output parameters do not need to be initialized before they passed to the method.
    - method must assign output parameters before exiting.
  - Reference parameters must be initialized before they are passed to the method.
    - Method passes a reference to an existing variable. If no initial value, that would be the equivalent of operating on an unassigned local variable.
- params
  - pass into a method a variable number of identically typed parameters (or classes related by inheritance) as a single logical parameter.
- Optional args
  - A parameter with a default value set by equal sign
  - If parameter not passed, default value is used
- Named args
  - Useful to avoid positional aspect of parameters
  - Useful when overloading
- Example
  - **OptionalParams**
  - **FunWithMethods**

```
class Program
   {
       static void Main(string[] args)
       {
           Console.WriteLine(Add(5, 7));
           Console.WriteLine(Add(5));
           Console.WriteLine(Add(y: 3, x: 8));
           Console.ReadLine();
       }
       static int Add(int x, int y = 10)
       {
           return x + y;
       }
   }
```

Method Overloading

- Unique arguments provide a "signature" for a method.
- Compiler can determine with method to invoke by the argument signature
- Example
  - **MethodOverloading**

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Method Overloading *****\n");

    // Calls int version of Add()
    Console.WriteLine(Add(10, 10));

    // Calls long version of Add()
    Console.WriteLine(Add(900000000000, 900000000000));

    // Calls double version of Add()
    Console.WriteLine(Add(4.3, 4.4));

    Console.ReadLine();
}

#region Overloaded Add() methods
// Overloaded Add() method.
static int Add(int x, int y)
{ Console.WriteLine("int method"); return x + y; }

static double Add(double x, double y)
{ Console.WriteLine("double method"); return x + y; }

static long Add(long x, long y)
{ Console.WriteLine("long method"); return x + y; }
#endregion
```

Arrays

- Simple
  - **`type[] varname`** – reference to a single dimensional array
    - `int[] numbers;`
    - Does not allocate space/resources
  - **`varname = new type[n]`** – allocates resources to a reference where n is size of array.
    - `numbers = new int[10];`
    - varname should have already been declared. Allocates an array of size 10.
  - Put it all together
    - **`type[] varname = new type[n]`**
    - `int[] numbers = new int[10];`
- type can be ANY object
  - object[] myObjects = new object[4]
  - Represents an array size 4 of the base class
  - References! Need a constructor unless basic type.
- Example
  - **FunWithArrays**

Array Initialization

- Use of {}, including comma separated values

```
// Array initialization syntax using the new keyword.
string[] stringArray = new string[] { "one", "two", "three" };
Console.WriteLine("stringArray has {0} elements", stringArray.Length);

// Array initialization syntax without using the new keyword.
bool[] boolArray = { false, false, true };
Console.WriteLine("boolArray has {0} elements", boolArray.Length);

// Array initialization with new keyword and size.
int[] intArray = new int[4] { 20, 22, 23, 0 };
Console.WriteLine("intArray has {0} elements", intArray.Length);
Console.WriteLine();
```

Implicit typed Arrays

- Use var keyword
- CANNOT mix value types though!

```
Console.WriteLine("=> Implicit Array Initialization.");

// a is really int[].
var a = new[] { 1, 10, 100, 1000 };
Console.WriteLine("a is a: {0}", a.ToString());

// b is really double[].
var b = new[] { 1, 1.5, 2, 2.5 };
Console.WriteLine("b is a: {0}", b.ToString());

// c is really string[].
var c = new[] { "hello", null, "world" };
Console.WriteLine("c is a: {0}", c.ToString());
```

Multidimensional Arrays

- Rectangular: new int[n,m]
- Jagged: new int[n][] – an array of different sized arrays.

```
static void RectMultidimensionalArray()
        {

            // A rectangular MD array.
            int[,] myMatrix;
            myMatrix = new int[3, 4];

            // Populate (3 * 4) array.
            for (int i = 0; i < 3; i++)
                for (int j = 0; j < 4; j++)
                    myMatrix[i, j] = i * j;

            // Print (3 * 4) array.
            for (int i = 0; i < 3; i++)
            {
                for (int j = 0; j < 4; j++)
                    Console.Write(myMatrix[i, j] + "\t");
                Console.WriteLine();
            }
            Console.WriteLine();
        }
```

```
static void JaggedMultidimensionalArray()
        {
            // A jagged MD array (i.e., an array of arrays).
            // Here we have an array of 5 different arrays.
            int[][] myJagArray = new int[5][];

            // Create the jagged array.
            for (int i = 0; i < myJagArray.Length; i++)
                myJagArray[i] = new int[i + 7];

            // Print each row (remember,
            // each element is defaulted to zero!)
            for (int i = 0; i < 5; i++)
            {
                for (int j = 0; j < myJagArray[i].Length; j++)
                    Console.Write(myJagArray[i][j] + " ");
                Console.WriteLine();
            }
            Console.WriteLine();
        }
```

Params, return values, and methods

- Arrays can be used as params or return values.
- Array base class has useful methods.

Table 4-2. *Select Members of System.Array*

Member of Array Class	Meaning in Life
`Clear()`	This static method sets a range of elements in the array to empty values (0 for numbers, `null` for object references, `false` for Booleans).
`CopyTo()`	This method is used to copy elements from the source array into the destination array.
`Length`	This property returns the number of items within the array.
`Rank`	This property returns the number of dimensions of the current array.
`Reverse()`	This static method reverses the contents of a one-dimensional array.
`Sort()`	This static method sorts a one-dimensional array of intrinsic types. If the elements in the array implement the `IComparer` interface, you can also sort your custom types (see Chapter 9).

enum

- set of symbolic names that map to known numerical values
- enum TypeName : integertype
  - Where integertype is byte, int, short, long, and specifies storage
  - Usually this is not needed.
- Can specify values (or not, will default to beginning with 0), or a beginning value with first member.
- Methods allow to get the string associated with an enum type
- Example
  - **FunWithEnums**

```
 // This time, EmpType maps to an underlying
byte.
    enum EmpType : byte
    {
        Manager = 10,
        Grunt = 1,
        Contractor = 100,
        VicePresident = 9
    }
```

Structures

- More generalized enums, or lightweight classes
- Define a type consisting of other fields (defined by their type)
  - The following is a new type consisting of two integers.
    - `struct MyType { public int x; public int y;}`
  - `You can also define methods and constructors`
- What makes a `struct` special?
  - Can be copied
  - Allocated on the stack
  - No need to use new but can if want to invoke a custom constructor
  - **Not a reference**
- Example
  - **FunWithStructures**

Call by Reference or Value

- If a reference type is passed by **reference**, the callee may change the values of the object's state data, as well as the object it is referencing.
- If a reference type is passed by **value**, the callee may change the values of the object's state data but *not* the object it is referencing.
  - This is the norm for objects

Final Details Regarding Value Types and Reference Types

To wrap up this topic, consider the information in Table 4-3, which summarizes the core distinctions between value types and reference types.

Table 4-3. *Value Types and Reference Types Comparison*

Intriguing Question	Value Type	Reference Type
Where are objects allocated?	Allocated on the stack.	Allocated on the managed heap.
How is a variable represented?	Value type variables are local copies.	Reference type variables are pointing to the memory occupied by the allocated instance.
What is the base type?	Implicitly extends `System.ValueType`.	Can derive from any other type (except `System. ValueType`), as long as that type is not "sealed" (more details on this in Chapter 6).
Can this type function as a base to other types?	No. Value types are always sealed and cannot be inherited from.	Yes. If the type is not sealed, it may function as a base to other types.

Table 4-3. (*continued*)

Intriguing Question	Value Type	Reference Type
What is the default parameter passing behavior?	Variables are passed by value (i.e., a copy of the variable is passed into the called function).	For reference types, the reference is copied by value.
Can this type override `System.Object.Finalize()`?	No.	Yes, indirectly (more details on this in Chapter 13).
Can I define constructors for this type?	Yes, but the default constructor is reserved (i.e., your custom constructors must all have arguments).	But, of course!
When do variables of this type die?	When they fall out of the defining scope.	When the object is garbage collected.

Despite their differences, value types and reference types both have the ability to implement interfaces and may support any number of fields, methods, overloaded operators, constants, properties, and events.

Nullable types

- A nullable type can represent all the values of its underlying type, plus the value null.
- Thus, if you declare a nullable `bool`, it could be assigned a value from the set {true, false, null}.
- Syntax: type? Indicates a nullable type
  - int? x
    - x can be any integer OR null (undefined).
- Useful with databases where a value can be null.
- Null coalescing operator is ??, acts like ?
  - int y? = x ?? 30;
- Example
  - **NullableTypes**