Methods and Parameter Modifiers

Methods can be implemented within the scope of classes or structures (as well as prototyped within interface types) and may be decorated with various keywords (e.g., static, virtual, public, new) to qualify their behavior.

Return Values and Expression Bodied Members

```
static int Add(int x, int y) => x + y;
```

This is syntactic sugar, meaning that the generated IL is no different. It's just another way to write the method. Some find it easier to read, and others don't, so the choice is yours which style you prefer.

The default manner in which a parameter is sent into a function is by value. If you do not mark an argument with a parameter modifier, a copy of the data is passed into the function.

Parameter Modifiers

- **(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.

The Default by Value Parameter

```
static int Add(int x, int y)
{
    int ans = x + y;
    // Caller will not see these changes  as you are modifying
    // a copy of the  original data.
    x = 10000;
    y = 88888;
    return ans;
}

static void Main(string[] args)
{
    // Pass two variables in by value.
    int x = 9, y = 10;
    Console.WriteLine($"Before call: X: {x}, Y: {y}");
    Console.WriteLine($"Answer is: { Add(x, y )}");
    Console.WriteLine($"After call: X: {x}, Y: {y}");
    Console.ReadLine();
}
```

Output will be

```
Before call: X: 9, Y: 10
Answer is: 19
After call: X: 9, Y: 10
```

The out Modifier

Methods that have been defined to take *output parameters* (via the **out** keyword) are under obligation to assign them to an appropriate value before exiting the method scope (if you fail to do so, you will receive compiler errors).

```
static void Add(int x, int y, out int ans)
{
        ans = x + y;
}
static void FillTheseValues(out int a, out string b, out bool c)
{
        a = 9;
        b = "Enjoy your string.";
        c = true;
}
```

Calling a method with output parameters also requires the use of the out modifier. However, the local variables that are passed as output variables are not required to be assigned before passing them in as output arguments (if you do so, the original value is lost after the call). The reason the compiler allows you to send in seemingly unassigned data is because the method being called must make an assignment.

```
static void Main(string[] args)
{
        Add(90, 90, out int ans);
        Console.WriteLine(ans);

        int i;
        string str;
        bool b;
        FillTheseValues(out i, out str, out b);
        Console.WriteLine($"{i}, {str}, {b}");

        Console.ReadLine();
}
```

The ref Modifier

The main difference between output and reference parameters is:

- Output parameters do not need to be initialized before they are passed to the method. The reason for this is that the method must assign output parameters before exiting.
- Reference parameters must be initialized before they are passed to the method. The reason for this is that you are passing a reference to an existing variable. If you don't assign it to an initial value, that would be the equivalent of operating on an unassigned local variable.

```
public static void SwapStrings(ref string s1, ref string s2)
{
        string tempStr = s1;
        s1 = s2;
        s2 = tempStr;
}
```

```
static void Main(string[] args)
{
        string str1 = "Flip";
        string str2 = "Flop";
        SwapStrings(ref str1, ref str2);
        Console.WriteLine(str1);   //  Flop
        Console.WriteLine(str2);   // Flip

        Console.ReadLine();
}
```

There are some rules around the ref keyword:

- Standard method results cannot be assigned to a ref local variable. The method must have been created as a ref return method.
- A local variable inside the ref method can't be returned as a ref local variable. The following code does not work:

```
ThisWillNotWork(string[] array)
{
        int foo = 5;
        return ref foo;
}
```

- This new feature doesn't work with async method.

The params Modifier

The **params** keyword allows you to pass into a method a variable number of identically typed parameters (or classes related by inheritance) as a single logical parameter. As well, arguments marked with the params keyword can be processed if the caller sends in a strongly typed array or a comma-delimited list of items.

```
// Return average of "some number" of doubles.
static double CalculateAverage(params double[] values)
{
        Console.WriteLine($"You sent me {values.Length } doubles.");

        double sum = 0;
        if(values.Length == 0)
        {
                return sum;
        }
        for (int i = 0; i < values.Length; i++)
        {
                sum += values[i];
        }
        return (sum / values.Length);
}
```

```
static void Main(string[] args)
{
        // Pass in a comma-delimited list of doubles...
        double average;
        average = CalculateAverage(4.0, 3.2, 5.7, 64.22, 87.2);
        Console.WriteLine($"Average of data is: { average }");

        // ...or pass an array of doubles.
        double[] data = { 4.0, 3.2, 5.7 };
        average = CalculateAverage(data);
        Console.WriteLine($"Average of data is: { average }");

        // Average of 0 is 0!
        Console.WriteLine($"Average of data is: { CalculateAverage()}");
        Console.ReadLine();
}
```

Invoking Methods Using Named Parameters

Named arguments allow you to invoke a method by specifying parameter values in any order you choose. Thus, rather than passing parameters solely by position (as you will do in most cases), you can choose to specify each argument by name using a colon operator.

If you begin to invoke a method using positional parameters, you must list them before any named parameters. In other words, named arguments must always be packed onto the end of a method call.

```
static void MethodForNamedArguments(ConsoleColor textColor,  ConsoleColor backgroundColor, int
age, string name)
{
        // Store old colors to restore after message is printed.
        ConsoleColor oldTextColor = Console.ForegroundColor;
        ConsoleColor oldbackgroundColor = Console.BackgroundColor;

        // Set new colors and print message.
        Console.ForegroundColor = textColor;
        Console.BackgroundColor = backgroundColor;

        Console.WriteLine($"{name}'s age is {age}");

        // Restore previous colors.
        Console.ForegroundColor = oldTextColor;
        Console.BackgroundColor = oldbackgroundColor;
}
static void Main(string[] args)
{
        MethodForNamedArguments (backgroundColor: ConsoleColor.Green, textColor:
        ConsoleColor.DarkBlue, name: "Leon", age: 15);

        Console.ReadLine();
}
```

Given that each argument has a default value, named arguments allow the caller to specify only the parameters for which they do not want to receive the defaults.

MethodForNamedArguments (name: "Helen");

Method Overloading

Like other modern object-oriented languages, C# allows a **method** to be **overloaded**. Simply put, when you define a set of identically named methods that differ by the number (or type) of parameters, the method in question is said to be overloaded.

```
static int Sum(int x, int y)
{
        return x + y;
}
static double Sum(double x, double y)
{
        return x + y;
}
static long Sum(long x, long y)
{
        return x + y;
}

Console.WriteLine(Sum(5, 36));
Console.WriteLine(Sum(5.2, 36.5));
```

Local Functions

A **local function** is a function declared inside another function. This new feature allows you to do the validation first and then encapsulate the real goal of the local function defined inside the method. This feature was added into the C# specification for custom iterator methods and asynchronous methods.

```
static int AddWrapper(int x, int y)
{
        //Do some validation here
        return Add();
        int Add()
        {
                return x + y;
        }
}
```