Defining and Calling Methods

What are Methods?

So far in this tutorial, you've only been working within one method of your own, the `Main` method of a console application. However, you've called many other methods in the course of completing the lessons so far and the associated exercises. A method is a function that is associated with a class. A function is a named series of statements in a program. Functions (and therefore methods) can optionally accept parameters, and they can optionally return a result. In this lesson, you'll see methods and functions generally referred to interchangeably.

Declaring Methods

There are two main kinds of methods in C#: *static* methods and *instance* methods. A static method is global to the program, and is called off of the type it is associated with. You learned about these kinds of methods in the Hello World lesson, since the `Main` method you've seen so much of is static. Instance methods are attached to an object instance. They can only be called if they are invoked from a non-null instance of an object of the appropriate type. Two objects of the same type will run the same code for a given instance method, but each method will have access to that object's internal state, so the results may be very different.

If you're writing a method that will only operate on the parameters being passed into it, you can typically declare it as a `static` method. Otherwise, your methods should be instance methods (which is the default if you don't add the `static` keyword).

```
static int Add(int operand1, int operand2)
{
    // only works with its own parameters; works well as static method
    return operand1 + operand2;
}
string FullName()
{
    // FirstName and LastName are Properties of this class - shouldn't be static
    return $"{FirstName} {LastName}";
}
```

Naming Methods

Of course, naming things is one of the hardest problems in computer science. It's worth putting a little thought into what you name methods, as they're one of the key building blocks you'll use to build larger software applications. Methods *do* things, so typically their names should include verbs that describe the action they perform. Methods should be small and focused. If you give the method a name that describes what it does, and the name includes "and", the method might be larger than it needs to be. Most of the time, methods should do one thing. However, if you do have methods that do multiple things, it's better to have a long, descriptive name than to give it a shorter, less descriptive method that doesn't cover everything it does. Your methods shouldn't surprise programmers who call them with unexpected behavior.

Frequently, you can create methods within your program by taking existing code (for instance, from your `Main` method), and moving it into a method, and replacing the original code with a call to the new method. This is especially important when you find that you're writing the same code in multiple places - you should try to follow the Don't Repeat Yourself (DRY) principle in your code whenever you can. However, it can also be useful even for code that only exists in one place, as a way to help keep your code modular and easy to understand. If you have a part of your program that is getting long and complex, to the point where you want to start adding comments that describe what different sections are doing, it's almost always a better choice to move blocks of codes into their own methods, and give the methods good names that make it clear what's being done in them.

As an example, consider this long block of code, which could appear in the program's `Main` method:

```
// display header
Console.WriteLine("----------------------------------------");
Console.WriteLine("**** My Super Program ******************");
Console.WriteLine("----------------------------------------");

// display the menu
Console.WriteLine("What do you want to do?");
Console.WriteLine("1 - View the Menu");
Console.WriteLine("2 - Exit the Program");

// additional code omitted
```

```
using System;

public class Program
{
    public static void Main()
    {
        // display header
        Console.WriteLine("----------------------------------------");
        Console.WriteLine("**** My Super Program ******************");
        Console.WriteLine("----------------------------------------");

        // display the menu
        Console.WriteLine("What do you want to do?");
        Console.WriteLine("1 - View the Menu");
```

```
        Console.WriteLine("2 - Exit the Program");

        // additional code omitted
    }
}
```

The above code could be greatly simplified by using methods:

```
DisplayHeader();
DisplayMenu();

switch (ReadCommand())
// additional code omitted



using System;

public class Program
{
    public static void Main()
    {
        DisplayHeader();
        DisplayMenu();

        // additional code omitted

    }

    static void DisplayHeader()
    {
        Console.WriteLine("----------------------------------------");
        Console.WriteLine("**** My Super Program ******************");
        Console.WriteLine("----------------------------------------");
    }

    static void DisplayMenu()
    {
        Console.WriteLine("What do you want to do?");
        Console.WriteLine("1 - View the Menu");
        Console.WriteLine("2 - Exit the Program");
    }
}
```

The two methods do not return anything; they simply contain the `Console.WriteLine` statements that previously were in the main method. You'll learn how to return results from methods in the next section.

Return Types

Methods are either declared as `void`, meaning they don't return a result, or they must declare a type they will return. You can see two examples of simple methods below:

```
static string CreateGreeting(string name)
{
    return $"Hi {name}!";
}

void DisplayGreeting()
{
    Console.WriteLine(CreateGreeting("Steve"));
}
```

```
using System;

public class Program
{
    public static void Main()
    {
        DisplayGreeting();
    }

    static string CreateGreeting(string name)
    {
        return $"Hi {name}!";
    }

    static void DisplayGreeting()
    {
        Console.WriteLine(CreateGreeting("Steve"));
    }
}
```

The first method returns a `string` type; the second one simply performs some action, without returning anything.

Methods can only return a single result, which sometimes presents a challenge when designing how a program will work, especially when considering edge cases. For example, some programs might return an integer representing the ID of a record that was created, but return a negative number to represent an error code. This kind of program design, which changes the meaning of a return type based on what its value might be based on some convention, is extremely error-prone. In .NET programming, it's recommended you avoid returning error codes from your methods, and instead raise *exceptions* if necessary when errors occur.

At other times, a method may need to return more than one value in certain circumstances. For example, parsing a string into a type might be an operation for which failure is not an error, but instead should just be represented by a `false` return type. However, when the parsing operation is successful, a value of `true` should be returned, along with the actual type created by the parsing operation. One approach to this problem is to use a parameter than can return the value (called an *out parameter*). You'll learn more about these in the next section.

Frequently, methods that need to return complex results should be modified in one of two ways. Either the type being returned should be replaced with a class that can encapsulate all of the information the operation needs to provide as a result, or the method itself should be moved into a class that can contain this behavior. Keep in mind when you're deciding how to return results from your methods that sometimes it will make sense to create a new class whose only role is to represent the result of a particular operation (often such classes are named with *Result* as a suffix). In most cases, it's better to use a result type that can accurately and fully describe the result of a function's action than to use out parameters for this same purpose.

Parameters

Methods can accept parameters, which can be used to modify their behavior. Parameters are declared inside of the parentheses in the method's declaration, as you've seen. Each parameter must specify a type and a name. C# supports optional parameters which will use default values when omitted. You can even create methods that take an arbitrary number of parameters by using a params array, but some of these behaviors are more advanced than you need to know for this lesson.

> **Tip** {.tip .java}
> The params array works the same way that *varargs* do in Java. Instead of `sum(int... list)`, just use `Sum(params int[] list)`.

The name of a method, combined with the types of its parameters, must be unique within the class to which the method belongs. This is known as as the method's *signature*. You will see an error when you build your program if you have two methods with the same name and set of parameter types, even if the return types are different or the parameters have different names. For example, having the following two methods within the same class would generate an error at compile time:

```
string GetValue(string fileName)
{}
int GetValue(string versionNumber)
{}
```

```
using System;

public class Program
{
    public static void Main()
    {
        // this program won't compile
    }

    string GetValue(string fileName)
    {
        return "";
    }

    int GetValue(string versionNumber)
    {
```

```
        return 0;
    }
}
```

It is possible to repeat the same name for a method, but when doing so each *signature*must be unique. These are referred to as *method overloads* and you'll learn more about this technique in the next section.

It's a good idea to limit the number of parameters a method accepts, since a large number of parameters often indicates a method is doing too much and should be broken up into smaller methods. As with the recommendation to use specialized objects for *Result* types, you may sometimes find that it makes sense to create objects to represent parameters as well, or even to create a new type for a method to live on that eliminates the need to pass it many parameters (in such cases, the type's properties would typically be used instead of parameters).

Parameters can be made optional by supplying them with default values. The syntax for this is simply to supply the value with the parameter declaration as an assignment:

```
static string CreateGreeting(string name = "You")
{
    return $"Hi {name}!";
}
```

```
using System;

public class Program
{
    public static void Main()
    {
        string greeting = CreateGreeting();
        Console.WriteLine($"Default Greeting: {greeting}");

        string customGreeting = CreateGreeting("Steve");
        Console.WriteLine($"Custom Greeting: {customGreeting}");
    }

    static string CreateGreeting(string name = "You")
    {
        return $"Hi {name}!";
    }
}
```

The above method can now be called with or without a `string` argument for the `name` parameter. If no argument is passed, the method will use "You" as the value for `name` . Note that required parameters must be declared before any optional parameters.

Overloads

C# has always supported the concept of method overloads, which are multiple declarations of the same named method, with different sets of parameters. In many cases, this feature was used to support cases that are now better supported by optional parameters (a more recent C# language feature). In any case, you can create method overloads to provide a richer interface for programmers, allowing them to choose the method signature that is best-suited to their needs. Frequently, method overloads are written such that the real work of the method is performed by the method with the largest number of parameters, and each overload will call this method, supplying the appropriate parameters. This helps to follow the DRY principle - you should avoid creating method overloads that simply copy behavior between themselves.

The following example demonstrates how method overloads might be used for a case where default parameters might also work:

```
static string CreateGreeting()
{
    // call version with more parameters, passing a default value
    return CreateGreeting("You");
}
string CreateGreeting(string name)
{
    return $"Hi {name}!";
}
```

```
using System;

public class Program
{
    public static void Main()
    {
        string greeting = CreateGreeting();
        Console.WriteLine($"Default Greeting: {greeting}");

        string customGreeting = CreateGreeting("Steve");
        Console.WriteLine($"Custom Greeting: {customGreeting}");
    }

    static string CreateGreeting()
    {
        // call version with more parameters, passing a default value
        return CreateGreeting("You");
    }

    static string CreateGreeting(string name)
    {
        return $"Hi {name}!";
    }
}
```

Sometimes, you want to help programmers who might use your method by eliminating the need for them to convert whatever type they have into the type your method expects as a parameter. Thus, you create multiple overloads that will perform the type conversion internally. This makes your method easier to use, and reduces the complexity of the code that calls it. For example, consider these methods that will display how many seconds remain until a deadline, or based on a given `TimeSpan`:

```
static int SecondsRemaining(DateTime endTime)
{
    return SecondsRemaining(endTime - DateTime.Now);
}

static int SecondsRemaining(string endTime)
{
    return SecondsRemaining(DateTime.Parse(endTime));
}

static int SecondsRemaining(TimeSpan duration)
{
    return (int)duration.TotalSeconds;
}
```

```
using System;

public class Program
{
    public static void Main()
    {
        DateTime end = DateTime.Now.AddSeconds(10);
        string endTimeString = end.ToString();
        TimeSpan timeLeft = end - DateTime.Now;

        Console.WriteLine($"Seconds: {SecondsRemaining(end)}");
        Console.WriteLine($"Seconds: {SecondsRemaining(endTimeString)}");
        Console.WriteLine($"Seconds: {SecondsRemaining(timeLeft)}");
    }

    static int SecondsRemaining(DateTime endTime)
    {
        return SecondsRemaining(endTime - DateTime.Now);
    }

    static int SecondsRemaining(string endTime)
    {
        return SecondsRemaining(DateTime.Parse(endTime));
    }

    static int SecondsRemaining(TimeSpan duration)
    {
        return (int)duration.TotalSeconds;
    }
}
```

Ultimately, all of the methods end up calling the last one, which takes a `TimeSpan`, but they perform different calculations and/or conversions along the way to make working with the method easier for the calling code. This kind of simplification is a key benefit your programs can get from the method overloading feature in C#.

Tip {.tip .newLanguage}
Remember to think about how you or other developers will use the methods you write. Try to write them so they're simple and easy to use and understand.

Lambda Expressions

So far in this lesson, you've created methods as members of classes. While those comprise the majority of all methods you'll see in C#, you will also want to learn about *lambda expressions*, a very special type of method.

Lambda expressions are a type of method created in-line in your code. Most often, you will pass them as a parameter to other methods. In C#, you're allowed to store methods inside of variables, however, that is not covered in detail here. This lesson only covers the basics, so please consider this example expression:

```
public static void Main()
{
    Func<int, int> addOne = x => x + 1; // this is the lambda expression
    Console.WriteLine(addOne(4));
}
```

```
using System;

public class Program
{
    public static void Main()
    {
        Func<int, int> addOne = x => x + 1; // this is the lambda expression
        Console.WriteLine(addOne(4));
    }
}
```

The expression in this example is being assigned to a variable of type `Func<int, int>`. Everything to the right of the `=` is the lambda expression, which has two parts. The (`=>`) is a special operator for lambda expressions splitting the method parameters from the code to execute. In the example above, on the second line when addOne is called, the 4 is passed into the lambda expression and assigned to `x`. Since the lambda expression is only one line, the result of that one line is the return value (there is no need for an explicit return statement).

The following are some examples of lambda expressions:

```
public static void Main()
{
    const int four = 4;
    Func<int, int> addOne = x => x + 1;
    Func<int, int, int> calcArea = (x,y) => x * y; // two parameters
    Func<int> twentyFive = () => calcArea(addOne(four), addOne(four)); // no parameters
    Console.WriteLine(twentyFive());
}
```

```
using System;

public class Program
{
    public static void Main()
    {
        const int four = 4;
        Func<int, int> addOne = x => x + 1;
        Func<int, int, int> calcArea = (x,y) => x * y; // two parameters
        Func<int> twentyFive = () => calcArea(addOne(four), addOne(four)); // no
parameters
        Console.WriteLine(twentyFive());
    }
}
```

Notice that the expression method, `twentyFive`, is able to use the other variables in its expression.

Lambda expressions can be very useful in describing simple functions. Two ways in which they are commonly used is for *predicates* and *selectors*, which return a bool and an object respectively. You'll see these in action in Introducing LINQ, the lesson covering language integrated queries (LINQ).

Extension Methods

There is a special kind of static method which allows you to extend your existing type with new functionality without modifying the class itself. This is often useful when dealing with classes provided by other assemblies. These types of methods are called *Extension Methods*, and you'll get to see them used extensively in the Introducing LINQlesson. Extension Methods are *static* methods of one class that behave like *instance*methods of another class. The key difference between extension methods and other static methods is using a special `this` parameter whose type is the class to be extended. In the following example, an extension method is created that will add five to the value of an integer and return the result:

```
public static class ExtensionMethods
{
    public static int PlusFive(this int input)
    {
        return input + 5;
    }
}
```

```
using System;

public class Program
{
    public static void Main()
    {
        int ten = 5.PlusFive();
        Console.WriteLine(ten);
    }
}

public static class ExtensionMethods
{
    public static int PlusFive(this int input)
    {
        return input + 5;
    }
}
```

Since the initial `int` parameter of the static method is preceded by the `this` modifier, you know this method is an extension method. Since the parameter is an `int`, it means the code may be used as a method on any `int` variable (or literal) you have.

```
int luckyNumber = 10;
Console.WriteLine(luckyNumber); // Will output 10
int result = luckyNumber.PlusFive();
Console.WriteLine(result); // Will output 15
```

```
using System;

public class Program
{
    public static void Main()
    {
        int luckyNumber = 10;
        Console.WriteLine(luckyNumber); // Will output 10
        int result = luckyNumber.PlusFive();
        Console.WriteLine(result); // Will output 15
    }
}

public static class ExtensionMethods
{
    public static int PlusFive(this int input)
    {
        return input + 5;
    }
}
```

If you attempt to use this extension method on a non-integer type, you will receive an error when compiling. For example, trying to call this `extension method` on a `string` type will cause the following.

```
error CS1929: 'string' does not contain a definition for 'PlusFive' and the best
extension method overload 'ExtensionMethods.PlusFive(int)' requires a receiver of type
'int'
```

Tip {.note}
Before using an extension method, your code will require a `using` statement for the namespace of your static class containing the extension method (if different from where you're accessing the extension method). You will learn more about these in the lesson on Understanding Namespaces.