CHAPTER 9

Methods

Methods are reusable code blocks that will only execute when called.

Defining Methods

A method can be created inside a class by typing `void` followed by the method's name, a set of parentheses, and a code block. The `void` keyword means that the method will not return a value. The naming convention for methods is the same as for classes – a descriptive name with each word initially capitalized.

```
class MyApp
{
  void MyPrint()
  {
    System.Console.WriteLine("Hello World");
  }
}
```

All methods in C# must belong to a class, and they are the only place where statements may be executed. C# does not have global functions, which are methods defined outside of classes.

M. Olsson, *C# 8 Quick Syntax Reference*, https://doi.org/10.1007/978-1-4842-5577-3_9

Calling Methods

The previously defined method will print out a text message. To invoke (call) it, an instance of the `MyApp` class must first be created by using the `new` keyword. The dot operator is then used after the instance's name to access its members, which includes the `MyPrint` method.

```
class MyApp
{
  static void Main()
  {
    MyApp m = new MyApp();
    m.MyPrint(); // Hello World
  }
  void MyPrint()
  {
    System.Console.WriteLine("Hello World");
  }
}
```

Method Parameters

The parentheses that follow the method name are used to pass arguments to the method. To do this, the corresponding parameters must first be specified in the method definition in the form of a comma-separated list of declarations.

```
void MyPrint(string s1, string s2)
{
  System.Console.WriteLine(s1 + s2);
}
```

A method can be defined to take any number of arguments, and they can have any data types. Just ensure the method is called with the same types and number of arguments.

```
static void Main()
{
  MyApp m = new MyApp();
  m.MyPrint("Hello", " World"); // "Hello World"
}
```

To be precise, *parameters* appear in method definitions, while *arguments* appear in method calls. However, the two terms are sometimes used interchangeably.

Params Keyword

To take a variable number of arguments of a specific type, an array with the `params` modifier can be added as the last parameter in the list. Any extra parameters of the specified type that are passed to the method will automatically be stored in that array.

```
void MyPrint(params string[] s)
{
  foreach (string x in s)
    System.Console.WriteLine(x);
}
```

Method Overloading

It is possible to declare multiple methods with the same name as long as the parameters vary in type or number. This is called *method overloading* and can be seen in the implementation of the `System.Console.Write`

method, for example, which has 18 method definitions. It is a powerful feature that allows a method to handle a variety of arguments without the programmer needing to be aware of using different methods.

```
void MyPrint(string s)
{
  System.Console.WriteLine(s);
}
void MyPrint(int i)
{
  System.Console.WriteLine(i);
}
```

Optional Parameters

As of C# 4.0, parameters can be declared as optional by providing a default value for them in the method declaration. When the method is invoked, these optional arguments may be omitted to use the default values.

```
class MyApp
{
  void MySum(int i, int j = 0, int k = 0)
  {
    System.Console.WriteLine(1*i + 2*j + 3*k);
  }
  static void Main()
  {
    new MyApp().MySum(1, 2); // 5
  }
}
```

Named Arguments

C# 4.0 also introduced *named arguments,* which allow an argument to be passed using the name of its corresponding parameter. This feature complements optional parameters by enabling arguments to be passed out of order, instead of relying on their position in the parameter list. Therefore, any optional parameter can be specified without having to specify the value for every optional parameter before it.

```
static void Main()
{
  new MyApp().MySum(1, k: 2); // 7
}
```

Both optional and required parameters can be named, but the named arguments must be placed after the unnamed ones. This order restriction was loosened in C# 7.2, allowing named arguments to be followed by positional arguments provided that the named arguments are in the correct position.

```
static void Main()
{
  new MyApp().MySum(i: 2, 1); // 4
}
```

Named arguments are useful for improving code readability, by identifying what each argument represents.

Return Statement

A method can return a value. The `void` keyword is then replaced with the data type that the method will return, and the `return` keyword is added to the method body with an argument of the specified return type.

```
string GetPrint()
{
  return "Hello";
}
```

`Return` is a jump statement that causes the method to exit and return the value to the place where the method was called. For example, the `GetPrint` method can be passed as an argument to the `Write` method since the method evaluates to a string.

```
static void Main()
{
  MyApp m = new MyApp();
  System.Console.Write(m.GetPrint()); // "Hello World"
}
```

The `return` statement may also be used in `void` methods to exit before the end block is reached.

```
void MyMethod()
{
  return;
}
```

Value and Reference Types

There are two kinds of data types in C#: *value types* and *reference types*. Variables of value types directly contain their data, whereas variables of reference types hold references to their data. The reference types in C# include class, interface, array, and delegate types. The value types include the simple types, as well as the `struct`, `enum`, and nullable value types. Reference type variables are typically created using the `new` keyword, although that is not always necessary, as, for example, in the case of string objects.

A variable of a reference type is generally called an *object*, although strictly speaking the object is the data that the variable refers to. With reference types, multiple variables can reference the same object, and therefore operations performed through one variable will affect any other variables that reference the same object. In contrast, with value types, each variable will store its own value and operations on one will not affect another.

Pass by Value

When passing parameters of value type, only a local copy of the variable is passed. This means that if the copy is changed, it will not affect the original variable.

```
void Set(int i) { i = 10; }

static void Main()
{
  MyApp m = new MyApp();
  int x = 0; // value type
  m.Set(x); // pass value of x
  System.Console.Write(x); // 0
}
```

Pass by Reference

For reference data types, C# uses true pass by reference. This means that when a reference type is passed, it is not only possible to change its state but also to replace the entire object and have the change propagate back to the original object.

```
void Set(int[] i) { i = new int[] { 10 }; }

static void Main()
{
  MyApp m = new MyApp();
  int[] y = { 0 }; // reference type
  m.Set(y); // pass object reference
  System.Console.Write(y[0]); // 10
}
```

Ref Keyword

A variable of value type can be passed by reference by using the `ref` keyword, both in the caller and method declarations. This will cause the variable to be passed in by reference, and therefore changing it will update the original value.

```
void Set(ref int i) { i = 10; }

static void Main()
{
  MyApp m = new MyApp();
  int x = 0; // value type
  m.Set(ref x); // pass reference to value type
  System.Console.Write(x); // 10
}
```

Value types can be returned by reference starting with C# 7.0. The `ref` keyword is then added both before the return type and the return value. Bear in mind that the returned variable must have a lifetime that extends beyond the method's scope, so it cannot be a variable local to the method.

```
class MyClass
{
  public int myField = 5;
  public ref int GetField()
  {
    return ref myField;
  }
}
```

The caller can decide whether to retrieve the returned variable by value (as a copy) or by reference (as an alias). Note that when retrieving by reference, the `ref` keyword is used both before the method call and before the variable declaration.

```
class MyApp
{
  static void Main()
  {
    MyClass m = new MyClass();
    ref int myAlias = ref m.GetField(); // reference
    int myCopy = m.GetField(); // value copy
    myAlias = 10;
    System.Console.WriteLine(m.myField); // "10"
  }
}
```

Out Keyword

Sometimes you may want to pass an unassigned variable by reference and have it assigned in the method. However, using an unassigned local variable will give a compile-time error. For this situation, the `out` keyword can be used. It has the same function as `ref`, except that the compiler will allow use of the unassigned variable, and it will make sure the variable is assigned in the method.

```
void Set(out int i) { i = 10; }

static void Main()
{
  MyApp m = new MyApp();
  int x; // value type
  m.Set(out x); // pass reference to unset value type
  System.Console.Write(x); // 10
}
```

With C# 7.0, it became possible to declare `out` variables in the argument list of a method call. This feature allows the previous example to be simplified in the following manner.

```
static void Main()
{
  MyApp m = new MyApp();
  m.Set(out int x);
  System.Console.Write(x); // 10
}
```

Local Methods

Starting with C# 7.0, a method can be defined inside another method. This is useful for limiting the scope of a method, in cases when the method is only called by one other method. To illustrate, a nested method is used here to perform a countdown. Note that this nested method calls itself and is therefore called a *recursive method*.

```
class MyClass
{
  void CountDown()
  {
    int x = 10;
    Recursion(x);
    System.Console.WriteLine("Done");
    void Recursion(int i)
    {
      if (i <= 0) return;
      System.Console.WriteLine(i);
      System.Threading.Thread.Sleep(1000); // wait 1 second
      Recursion(i - 1);
    }
  }

  static void Main()
  {
    new MyClass().CountDown();
  }
}
```