Methods in C#

Article • 02/13/2023 • 21 minutes to read

A method is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments. In C#, every executed instruction is performed in the context of a method. The `Main` method is the entry point for every C# application and it's called by the common language runtime (CLR) when the program is started.

> **Note**
>
> This topic discusses named methods. For information about anonymous functions, see **Lambda expressions**.

Method signatures

Methods are declared in a `class`, `record`, or `struct` by specifying:

- An optional access level, such as `public` or `private`. The default is `private`.
- Optional modifiers such as `abstract` or `sealed`.
- The return value, or `void` if the method has none.
- The method name.
- Any method parameters. Method parameters are enclosed in parentheses and are separated by commas. Empty parentheses indicate that the method requires no parameters.

These parts together form the method signature.

> **Important**
>
> A return type of a method is not part of the signature of the method for the purposes of method overloading. However, it is part of the signature of the method when determining the compatibility between a delegate and the method that it points to.

The following example defines a class named `Motorcycle` that contains five methods:

C#

```
using System;

abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() {/* Method statements here */ }

    // Only derived classes can call this.
    protected void AddGas(int gallons) { /* Method statements here */ }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method statements
here */ return 1; }

    // Derived classes can override the base class implementation.
    public virtual int Drive(TimeSpan time, int speed) { /* Method state-
ments here */ return 0; }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}
```

The `Motorcycle` class includes an overloaded method, `Drive`. Two methods have the same name, but must be differentiated by their parameter types.

Method invocation

Methods can be either *instance* or *static*. Invoking an instance method requires that you instantiate an object and call the method on that object; an instance method operates on that instance and its data. You invoke a static method by referencing the name of the type to which the method belongs; static methods don't operate on instance data. Attempting to call a static method through an object instance generates a compiler error.

Calling a method is like accessing a field. After the object name (if you're calling an instance method) or the type name (if you're calling a `static` method), add a period, the name of the method, and parentheses. Arguments are listed within the parentheses and are separated by commas.

The method definition specifies the names and types of any parameters that are required. When a caller invokes the method, it provides concrete values, called arguments, for each parameter. The arguments must be compatible with the parameter type, but the argument name, if one is used in the calling code, doesn't have to be the same as the parameter named defined in the method. In the following example, the `Square` method includes a single parameter of type `int` named *i*. The first method call passes the `Square` method a variable of type `int` named *num*; the second, a numeric constant; and the third, an expression.

C#

```
public class SquareExample
{
   public static void Main()
   {
      // Call with an int variable.
      int num = 4;
      int productA = Square(num);

      // Call with an integer literal.
      int productB = Square(12);

      // Call with an expression that evaluates to int.
      int productC = Square(productA * 3);
   }

   static int Square(int i)
   {
      // Store input argument in a local variable.
      int input = i;
      return input * input;
   }
}
```

The most common form of method invocation used positional arguments; it supplies arguments in the same order as method parameters. The methods of the `Motorcycle` class can therefore be called as in the following example. The call to the `Drive` method, for example, includes two arguments that correspond to the two parameters in the method's syntax. The first becomes the value of the `miles` parameter, the second the value of the `speed` parameter.

C#

```
class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}
```

You can also use *named arguments* instead of positional arguments when invoking a method. When using named arguments, you specify the parameter name followed by a colon (":") and the argument. Arguments to the method can appear in any order, as long as all required arguments are present. The following example uses named arguments to invoke the `TestMotorcycle.Drive` method. In this example, the named arguments are passed in the opposite order from the method's parameter list.

C#

```
using System;

class TestMotorcycle : Motorcycle
{
    public override int Drive(int miles, int speed)
    {
        return (int)Math.Round(((double)miles) / speed, 0);
    }

    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {
```

```
        TestMotorcycle moto = new TestMotorcycle();
        moto.StartEngine();
        moto.AddGas(15);
        var travelTime = moto.Drive(speed: 60, miles: 170);
        Console.WriteLine("Travel time: approx. {0} hours", travelTime);
    }
}
// The example displays the following output:
//       Travel time: approx. 3 hours
```

You can invoke a method using both positional arguments and named arguments. However, positional arguments can only follow named arguments when the named arguments are in the correct positions. The following example invokes the `TestMotorcycle.Drive` method from the previous example using one positional argument and one named argument.

C#

```
var travelTime = moto.Drive(170, speed: 55);
```

Inherited and overridden methods

In addition to the members that are explicitly defined in a type, a type inherits members defined in its base classes. Since all types in the managed type system inherit directly or indirectly from the Object class, all types inherit its members, such as Equals(Object), GetType(), and ToString(). The following example defines a `Person` class, instantiates two `Person` objects, and calls the `Person.Equals` method to determine whether the two objects are equal. The `Equals` method, however, isn't defined in the `Person` class; it's inherited from Object.

C#

```
using System;

public class Person
{
   public String FirstName;
}

public class ClassTypeExample
{
```

```
    public static void Main()
    {
       var p1 = new Person();
       p1.FirstName = "John";
       var p2 = new Person();
       p2.FirstName = "John";
       Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}
// The example displays the following output:
//       p1 = p2: False
```

Types can override inherited members by using the `override` keyword and providing an implementation for the overridden method. The method signature must be the same as that of the overridden method. The following example is like the previous one, except that it overrides the Equals(Object) method. (It also overrides the GetHashCode() method, since the two methods are intended to provide consistent results.)

C#

```
using System;

public class Person
{
    public String FirstName;

    public override bool Equals(object obj)
    {
        var p2 = obj as Person;
        if (p2 == null)
            return false;
        else
            return FirstName.Equals(p2.FirstName);
    }

    public override int GetHashCode()
    {
        return FirstName.GetHashCode();
    }
}

public class Example
{
    public static void Main()
    {
        var p1 = new Person();
        p1.FirstName = "John";
```

```
        var p2 = new Person();
        p2.FirstName = "John";
        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}
// The example displays the following output:
//       p1 = p2: True
```

Passing parameters

Types in C# are either *value types* or *reference types*. For a list of built-in value types, see Types. By default, both value types and reference types are passed to a method by value.

Passing parameters by value

When a value type is passed to a method by value, a copy of the object instead of the object itself is passed to the method. Therefore, changes to the object in the called method have no effect on the original object when control returns to the caller.

The following example passes a value type to a method by value, and the called method attempts to change the value type's value. It defines a variable of type `int`, which is a value type, initializes its value to 20, and passes it to a method named `ModifyValue` that changes the variable's value to 30. When the method returns, however, the variable's value remains unchanged.

C#

```
using System;

public class ByValueExample
{
   public static void Main()
   {
      int value = 20;
      Console.WriteLine("In Main, value = {0}", value);
      ModifyValue(value);
      Console.WriteLine("Back in Main, value = {0}", value);
   }

   static void ModifyValue(int i)
   {
```

```
        i = 30;
        Console.WriteLine("In ModifyValue, parameter value = {0}", i);
        return;
    }
}
// The example displays the following output:
//       In Main, value = 20
//       In ModifyValue, parameter value = 30
//       Back in Main, value = 20
```

When an object of a reference type is passed to a method by value, a reference to the object is passed by value. That is, the method receives not the object itself, but an argument that indicates the location of the object. If you change a member of the object by using this reference, the change is reflected in the object when control returns to the calling method. However, replacing the object passed to the method has no effect on the original object when control returns to the caller.

The following example defines a class (which is a reference type) named `SampleRefType`. It instantiates a `SampleRefType` object, assigns 44 to its `value` field, and passes the object to the `ModifyObject` method. This example does essentially the same thing as the previous example—it passes an argument by value to a method. But because a reference type is used, the result is different. The modification that is made in `ModifyObject` to the `obj.value` field also changes the `value` field of the argument, `rt`, in the `Main` method to 33, as the output from the example shows.

C#

```
using System;

public class SampleRefType
{
    public int value;
}

public class ByRefTypeExample
{
    public static void Main()
    {
        var rt = new SampleRefType();
        rt.value = 44;
        ModifyObject(rt);
        Console.WriteLine(rt.value);
    }
```

```
    static void ModifyObject(SampleRefType obj)
    {
        obj.value = 33;
    }
}
```

Passing parameters by reference

You pass a parameter by reference when you want to change the value of an argument in a method and want to reflect that change when control returns to the calling method. To pass a parameter by reference, you use the ref or out keyword. You can also pass a value by reference to avoid copying but still prevent modifications using the in keyword.

The following example is identical to the previous one, except the value is passed by reference to the `ModifyValue` method. When the value of the parameter is modified in the `ModifyValue` method, the change in value is reflected when control returns to the caller.

C#

```
using System;

public class ByRefExample
{
   public static void Main()
   {
      int value = 20;
      Console.WriteLine("In Main, value = {0}", value);
      ModifyValue(ref value);
      Console.WriteLine("Back in Main, value = {0}", value);
   }

   static void ModifyValue(ref int i)
   {
      i = 30;
      Console.WriteLine("In ModifyValue, parameter value = {0}", i);
      return;
   }
}
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 30
```

A common pattern that uses by ref parameters involves swapping the values of variables. You pass two variables to a method by reference, and the method swaps their contents. The following example swaps integer values.

C#

```
using System;

public class RefSwapExample
{
   static void Main()
   {
      int i = 2, j = 3;
      System.Console.WriteLine("i = {0}  j = {1}" , i, j);

      Swap(ref i, ref j);

      System.Console.WriteLine("i = {0}  j = {1}" , i, j);
   }

   static void Swap(ref int x, ref int y)
   {
      int temp = x;
      x = y;
      y = temp;
   }
}
// The example displays the following output:
//      i = 2  j = 3
//      i = 3  j = 2
```

Passing a reference-type parameter allows you to change the value of the reference itself, rather than the value of its individual elements or fields.

Parameter arrays

Sometimes, the requirement that you specify the exact number of arguments to your method is restrictive. By using the `params` keyword to indicate that a parameter is a parameter array, you allow your method to be called with a variable number of arguments. The parameter tagged with the `params` keyword must be an array type, and it must be the last parameter in the method's parameter list.

A caller can then invoke the method in either of four ways:

- By passing an array of the appropriate type that contains the desired number of elements.
- By passing a comma-separated list of individual arguments of the appropriate type to the method.
- By passing `null`.
- By not providing an argument to the parameter array.

The following example defines a method named `GetVowels` that returns all the vowels from a parameter array. The `Main` method illustrates all four ways of invoking the method. Callers aren't required to supply any arguments for parameters that include the `params` modifier. In that case, the parameter is an empty array.

C#

```
using System;
using System.Linq;

class ParamsExample
{
    static void Main()
    {
        string fromArray = GetVowels(new[] { "apple", "banana", "pear" });
        Console.WriteLine($"Vowels from array: '{fromArray}'");

        string fromMultipleArguments = GetVowels("apple", "banana", "pear");
        Console.WriteLine($"Vowels from multiple arguments:
'{fromMultipleArguments}'");

        string fromNull = GetVowels(null);
        Console.WriteLine($"Vowels from null: '{fromNull}'");

        string fromNoValue = GetVowels();
        Console.WriteLine($"Vowels from no value: '{fromNoValue}'");
    }

    static string GetVowels(params string[] input)
    {
        if (input == null || input.Length == 0)
        {
            return string.Empty;
        }

        var vowels = new char[] { 'A', 'E', 'I', 'O', 'U' };
        return string.Concat(
            input.SelectMany(
                word => word.Where(letter => vow-
```

```
els.Contains(char.ToUpper(letter)))));
    }
}

// The example displays the following output:
//     Vowels from array: 'aeaaaea'
//     Vowels from multiple arguments: 'aeaaaea'
//     Vowels from null: ''
//     Vowels from no value: ''
```

Optional parameters and arguments

A method definition can specify that its parameters are required or that they're optional. By default, parameters are required. Optional parameters are specified by including the parameter's default value in the method definition. When the method is called, if no argument is supplied for an optional parameter, the default value is used instead.

The parameter's default value must be assigned by one of the following kinds of expressions:

- A constant, such as a literal string or number.
- An expression of the form `default(SomeType)`, where `SomeType` can be either a value type or a reference type. If it's a reference type, it's effectively the same as specifying `null`. You can use the `default` literal, as the compiler can infer the type from the parameter's declaration.
- An expression of the form `new ValType()`, where `ValType` is a value type. This invokes the value type's implicit parameterless constructor, which isn't an actual member of the type.

> **Note**
>
> In C# 10 and later, when an expression of the form `new ValType()` invokes the explicitly defined parameterless constructor of a value type, the compiler generates an error as the default parameter value must be a compile-time constant. Use the `default(ValType)` expression or the `default` literal to provide the default parameter value. For more information about parameterless constructors, see the **Struct initialization and default values**

section of the **Structure types** article.

If a method includes both required and optional parameters, optional parameters are defined at the end of the parameter list, after all required parameters.

The following example defines a method, `ExampleMethod`, that has one required and two optional parameters.

C#

```
using System;

public class Options
{
    public void ExampleMethod(int required, int optionalInt = default,
                              string? description = default)
   {
        var msg = $"{description ?? "N/A"}: {required} + {optionalInt} =
{required + optionalInt}";
        Console.WriteLine(msg);
   }
}
```

If a method with multiple optional arguments is invoked using positional arguments, the caller must supply an argument for all optional parameters from the first one to the last one for which an argument is supplied. In the case of the `ExampleMethod` method, for example, if the caller supplies an argument for the `description` parameter, it must also supply one for the `optionalInt` parameter. `opt.ExampleMethod(2, 2, "Addition of 2 and 2");` is a valid method call; `opt.ExampleMethod(2, , "Addition of 2 and 0");` generates an "Argument missing" compiler error.

If a method is called using named arguments or a combination of positional and named arguments, the caller can omit any arguments that follow the last positional argument in the method call.

The following example calls the `ExampleMethod` method three times. The first two method calls use positional arguments. The first omits both optional arguments, while the second omits the last argument. The third method call supplies a positional argument for the required parameter but uses a named argument to supply a value to the `description` parameter while omitting the `optionalInt` argument.

C#

```
public class OptionsExample
{
   public static void Main()
   {
      var opt = new Options();
      opt.ExampleMethod(10);
      opt.ExampleMethod(10, 2);
      opt.ExampleMethod(12, description: "Addition with zero:");
   }
}
// The example displays the following output:
//      N/A: 10 + 0 = 10
//      N/A: 10 + 2 = 12
//      Addition with zero:: 12 + 0 = 12
```

The use of optional parameters affects *overload resolution*, or the way in which the C# compiler determines which particular overload should be invoked by a method call, as follows:

- A method, indexer, or constructor is a candidate for execution if each of its parameters either is optional or corresponds, by name or by position, to a single argument in the calling statement, and that argument can be converted to the type of the parameter.
- If more than one candidate is found, overload resolution rules for preferred conversions are applied to the arguments that are explicitly specified. Omitted arguments for optional parameters are ignored.
- If two candidates are judged to be equally good, preference goes to a candidate that doesn't have optional parameters for which arguments were omitted in the call. This is a consequence of a general preference in overload resolution for candidates that have fewer parameters.

Return values

Methods can return a value to the caller. If the return type (the type listed before the method name) isn't `void`, the method can return the value by using the `return` keyword. A statement with the `return` keyword followed by a variable, constant, or expression that matches the return type will return that value to the method caller. Methods with a non-void return type are required to use the `return` keyword to return

a value. The `return` keyword also stops the execution of the method.

If the return type is `void`, a `return` statement without a value is still useful to stop the execution of the method. Without the `return` keyword, the method will stop executing when it reaches the end of the code block.

For example, these two methods use the `return` keyword to return integers:

C#

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

To use a value returned from a method, the calling method can use the method call itself anywhere a value of the same type would be sufficient. You can also assign the return value to a variable. For example, the following two code examples accomplish the same goal:

C#

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

C#

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

Using a local variable, in this case, `result`, to store a value is optional. It may help the

readability of the code, or it may be necessary if you need to store the original value of the argument for the entire scope of the method.

Sometimes, you want your method to return more than a single value. You can do this easily by using *tuple types* and *tuple literals*. The tuple type defines the data types of the tuple's elements. Tuple literals provide the actual values of the returned tuple. In the following example, `(string, string, string, int)` defines the tuple type that is returned by the `GetPersonalInfo` method. The expression `(per.FirstName, per.MiddleName, per.LastName, per.Age)` is the tuple literal; the method returns the first, middle, and last name, along with the age, of a `PersonInfo` object.

C#

```
public (string, string, string, int) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

The caller can then consume the returned tuple with code like the following:

C#

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.Item1} {person.Item3}: age = {person.Item4}");
```

Names can also be assigned to the tuple elements in the tuple type definition. The following example shows an alternate version of the `GetPersonalInfo` method that uses named elements:

C#

```
public (string FName, string MName, string LName, int Age)
GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

The previous call to the `GetPersonalInfo` method can then be modified as follows:

C#

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.FName} {person.LName}: age = {person.Age}");
```

If a method is passed an array as an argument and modifies the value of individual elements, it isn't necessary for the method to return the array, although you may choose to do so for good style or functional flow of values. This is because C# passes all reference types by value, and the value of an array reference is the pointer to the array. In the following example, changes to the contents of the `values` array that are made in the `DoubleValues` method are observable by any code that has a reference to the array.

C#

```
using System;

public class ArrayValueExample
{
   static void Main(string[] args)
   {
      int[] values = { 2, 4, 6, 8 };
      DoubleValues(values);
      foreach (var value in values)
         Console.Write("{0}  ", value);
   }

   public static void DoubleValues(int[] arr)
   {
      for (int ctr = 0; ctr <= arr.GetUpperBound(0); ctr++)
         arr[ctr] = arr[ctr] * 2;
   }
}
// The example displays the following output:
//       4  8  12  16
```

Extension methods

Ordinarily, there are two ways to add a method to an existing type:

- Modify the source code for that type. You can't do this, of course, if you don't own the type's source code. And this becomes a breaking change if you also add any

private data fields to support the method.
- Define the new method in a derived class. A method can't be added in this way using inheritance for other types, such as structures and enumerations. Nor can it be used to "add" a method to a sealed class.

Extension methods let you "add" a method to an existing type without modifying the type itself or implementing the new method in an inherited type. The extension method also doesn't have to reside in the same assembly as the type it extends. You call an extension method as if it were a defined member of a type.

For more information, see Extension Methods.

Async Methods

By using the async feature, you can invoke asynchronous methods without using explicit callbacks or manually splitting your code across multiple methods or lambda expressions.

If you mark a method with the async modifier, you can use the await operator in the method. When control reaches an `await` expression in the async method, control returns to the caller if the awaited task isn't completed, and progress in the method with the `await` keyword is suspended until the awaited task completes. When the task is complete, execution can resume in the method.

> **Note**
>
> An async method returns to the caller when either it encounters the first awaited object that's not yet complete or it gets to the end of the async method, whichever occurs first.

An async method typically has a return type of Task<TResult>, Task, IAsyncEnumerable<T> or `void`. The `void` return type is used primarily to define event handlers, where a `void` return type is required. An async method that returns `void` can't be awaited, and the caller of a void-returning method can't catch exceptions that the method throws. An async method can have any task-like return type.

In the following example, `DelayAsync` is an async method that has a return statement that returns an integer. Because it's an async method, its method declaration must have

a return type of `Task<int>`. Because the return type is `Task<int>`, the evaluation of the `await` expression in `DoSomethingAsync` produces an integer, as the following `int result = await delayTask` statement demonstrates.

C#

```
class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
//   Result: 5
```

An async method can't declare any in, ref, or out parameters, but it can call methods that have such parameters.

For more information about async methods, see Asynchronous programming with async and await and Async return types.

Expression-bodied members

It's common to have method definitions that simply return immediately with the result of an expression, or that have a single statement as the body of the method. There's a syntax shortcut for defining such methods using `=>`:

C#

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

If the method returns `void` or is an async method, the body of the method must be a statement expression (same as with lambdas). For properties and indexers, they must be read-only, and you don't use the `get` accessor keyword.

Iterators

An iterator performs a custom iteration over a collection, such as a list or an array. An iterator uses the yield return statement to return each element one at a time. When a `yield return` statement is reached, the current location is remembered so that the caller can request the next element in the sequence.

The return type of an iterator can be IEnumerable, IEnumerable<T>, IAsyncEnumerable<T>, IEnumerator, or IEnumerator<T>.

For more information, see Iterators.

See also

- Access Modifiers
- Static Classes and Static Class Members
- Inheritance
- Abstract and Sealed Classes and Class Members
- params
- out
- ref
- in
- Passing Parameters