200 XP

Exercise - Call the methods of a .NET Class

12 minutes

Whether you realized it or not, you've been calling C# methods ever since your first "Hello, World!" application. That application uses the `WriteLine()` method of the `Console` class to display the "Hello, World!" message.

However, not all classes and methods are implemented the same way. This unit covers some of the most common variants that you'll need to understand when using methods from the .NET Class Library. More importantly, you'll learn how to find and use the documentation to better understand more about each method.

How to call methods in the .NET Class Library

From your previous experience with the `Console.WriteLine()` method, you should already know the basics:

- Start by typing the class name. In this case, the class name is `Console`.
- Add the member access operator, the `.` symbol.
- Add the method's name. In this case, the method's name is `WriteLine`.
- Add the method invocation operator, which is a set of parentheses `()`.
- Finally, specify the arguments that are passed to the method, if there are any, between the parentheses of the method invocation operator. In this case, you specify the text that you want the `Console.WriteLine()` method to write to the console (for example, `"Hello World!"`).

Optionally, depending on how the developers designed and implemented the given method, you may also need to:

- Pass additional values as input parameters.
- Accept a return value.

In the next unit, you'll examine how to pass input values to a method, and how a method can be used to return a value to the calling routine.

While some methods can be called the same way that you called `Console.WriteLine()`, there are other methods in the .NET Class Library that require a different approach.

Prepare your coding environment

This module includes coding activities that guide you through the process of building and running sample code. You are encouraged to complete these activities using Visual Studio Code as your development environment. Using Visual Studio Code for these activities will help you to become more comfortable writing and running code in a developer environment that's used by professionals worldwide.

1. Open Visual Studio Code.

   You can use the Windows Start menu (or equivalent resource for another OS) to open Visual Studio Code.

2. On the Visual Studio Code **File** menu, select **Open Folder**.

3. In the **Open Folder** dialog, navigate to the **Windows Desktop** folder.

   If you have a different folder location where you keep code projects, you can use that folder location instead. For this training, the important thing is to have a location that's easy to locate and remember.

4. In the **Open Folder** dialog, select **Select Folder**.

   If you see a security dialog asking if you trust the authors, select **Yes**.

5. On the Visual Studio Code **Terminal** menu, select **New Terminal**.

   Notice that a command prompt in the Terminal panel displays the folder path for the current folder. For example:

   ```dos
   C:\Users\someuser\Desktop>
   ```

   > **Note**
   >
   > If you are working on your own PC rather than in a sandbox or hosted environment and you have completed other Microsoft Learn modules in this C# series, you may have

already created a project folder for code samples. If that's the case, you can skip past the next step, which is used to create a console app in the TestProject folder.

6. At the Terminal command prompt, to create a new console application in a specified folder, type **dotnet new console -o ./CsharpProjects/TestProject** and then press Enter.

   This .NET CLI command uses a .NET program template to create a new C# console application project in the specified folder location. The command creates the CsharpProjects and TestProject folders for you, and uses TestProject as the name of your `.csproj` file.

7. In the EXPLORER panel, expand the **CsharpProjects** folder.

   You should see the TestProject folder and two files, a C# program file named Program.cs and a C# project file named TestProject.csproj.

8. In the EXPLORER panel, to view your code file in the Editor panel, select **Program.cs**.

9. Delete the existing code lines.

   You'll be using this C# console project to create, build, and run code samples during this module.

10. Close the Terminal panel.

Call different kinds of methods in the .NET Class Library

1. In the Visual Studio Code Editor, to create a code sample that implements methods of the `System.Random` and `System.Console` classes, enter the following code:

   ```c#
   Random dice = new Random();
   int roll = dice.Next(1, 7);
   Console.WriteLine(roll);
   ```

   This code simulates a dice roll using the `Random.Next()` method to generate a number, and the `Console.WriteLine()` method to display the value.

> **Note**
>
> You will examine the code in detail later in this unit.

2. On the Visual Studio Code **File** menu, click **Save**.

3. In the EXPLORER panel, to open a Terminal at your TestProject folder location, right-click **TestProject**, and then select **Open in Integrated Terminal**.

   Notice that the Terminal panel includes a command prompt that displays a folder path. For example:

   `C:\Users\someuser\Desktop\CsharpProjects\TestProject>`

   When you use the Terminal to run .NET CLI commands, this folder location is where the commands will be running. Ensure that your code folder matches the folder path displayed in the command prompt before you build or run your code.

4. At the Terminal command prompt, to run your code, type **dotnet run** and then press Enter.

   Notice that a number from 1 to 6 is displayed in the console output (the number of dots on the dice). If you run the code enough times, you will eventually see each of the numbers displayed.

5. Take a minute to examine the syntax used to access the `Next()` and `WriteLine()` methods.

   Notice that you use different techniques to access the methods.

   ```
   c#

   Random dice = new Random();
   int roll = dice.Next(1, 7);
   Console.WriteLine(roll);
   ```

   On the third code line, you include a reference to the `Console` class and call the `Console.WriteLine()` method directly. However, you use a different technique for calling the `Random.Next()` method. The reason why you're using two different techniques is because some methods are stateful and others are stateless. Your next step is to examine the difference between stateful and stateless methods.

Stateful versus stateless methods

In software development projects, the term **state** is used to describe the condition of the execution environment at a specific moment in time. As your code executes line by line, values are stored in variables. At any moment during execution, the current state of the application is the collection of all values stored in memory.

Some methods don't rely on the current state of the application to work properly. In other words, **stateless methods** are implemented so that they can work without referencing or changing any values already stored in memory. Stateless methods are also known as **static methods**.

For example, the `Console.WriteLine()` method doesn't rely on any values stored in memory. It performs its function and finishes without impacting the state of the application in any way.

Other methods, however, must have access to the state of the application to work properly. In other words, **stateful methods** are built in such a way that they rely on values stored in memory by previous lines of code that have already been executed. Or they modify the state of the application by updating values or storing new values in memory. They're also known as **instance methods**.

Stateful (instance) methods keep track of their state in *fields*, which are variables defined on the class. Each new instance of the class gets its own copy of those fields in which to store state.

A single class can support both stateful and stateless methods. However, when you need to call stateful methods, you must first create an *instance* of the class so that the method can access state.

Creating an instance of a class

An instance of a class is called an *object*. To create a new instance of a class, you use the `new` operator. Consider the following line of code that creates a new instance of the `Random` class to create a new object called `dice`:

```
C#

Random dice = new Random();
```

The `new` operator does several important things:

- It first requests an address in the computer's memory large enough to store a new object based on the `Random` class.
- It creates the new object, and stores it at the memory address.
- It returns the memory address so that it can be saved in the `dice` variable.

From that point on, when the `dice` variable is referenced, the .NET Runtime performs a lookup behind the scenes to give the illusion that you're working directly with the object itself.

The latest version of the .NET Runtime enables you to instantiate an object without having to repeat the type name (target-typed constructor invocation). For example, the following code will create a new instance of the `Random` class:

```
c#

Random dice = new();
```

The intention is to simplify code readability. You always use parentheses when writing a target-typed `new` expression.

Why is the Next() method stateful?

You might be wondering why the `Next()` method was implemented as a stateful method? Couldn't the .NET Class Library designers figure out a way to generate a random number without requiring state? And what exactly is being stored or referenced by the `Next()` method?

These are fair questions. At a high level, computers are good at following specific instructions to create a reliable and repeatable outcome. To create the illusion of randomness, the developers of the `Next()` method decided to capture the date and time down to the fraction of a millisecond and use that to seed an algorithm that produces a different number each time. While not entirely random, it suffices for most applications. The state that is captured and maintained through the lifetime of the `dice` object is the seed value. Each subsequent call to the `Next()` method is rerunning the algorithm, but ensures that the seed changes so that the same value isn't (necessarily) returned.

To use the `Random.Next()` method, however, you don't have to understand *how* it works. The important thing to know is that some methods require you to create an instance of a class before you call them, while others do not.

How can you determine whether you need to create an instance of a class before calling its methods?

One approach for determining whether a method is stateful or stateless is to consult the documentation. The documentation includes examples that show whether the method must be called from the object instance or directly from the class.

> **Note**
>
> You may need to scroll down on the documentation page to find the code examples.

As an alternative to searching through product documentation, you can attempt to access the method directly from the class itself. If it works, you know that it's a stateless method. The worst that can happen is that you'll get a compilation error.

Try accessing the `Random.Next()` method directly and see what happens.

1. Enter the following line of code into the Visual Studio Code Editor:

   ```
   c#

   int result = Random.Next();
   ```

   You already know that `Next()` is a stateful method, however this example demonstrates how the Visual Studio Code Editor reacts when you try to access a method incorrectly.

2. Notice that a red squiggly line appears under `Random.Next`, indicating that you have a compilation error.

   If the method that you're interested in using is stateless, no red squiggly line will appear.

3. Hover your mouse pointer over the red squiggly line.

   A popup window should appear with the following message:

   ```
   Output

   (1,14): error CS0120: An object reference is required for the non-static field,
   method, or property 'Random.Next()'
   ```

As you saw in the code at the beginning of the unit, you can fix this error by creating an instance of the `Random` class before accessing the `Next()` method. For example:

```c#
Random dice = new Random();
int roll = dice.Next();
```

In this case, the `Next()` method is called without input parameters.

Recap

- To call methods of a class in the .NET Class Library, you use the format `ClassName.MethodName()`, where the `.` symbol is the member access operator to access a method defined on the class, and the `()` symbols are the method invocation operators.
- When calling a stateless method, you don't need to create a new instance of its class first.
- When calling a stateful method, you need to create an instance of the class, and access the method on the object.
- Use the `new` operator to create a new instance of a class.
- An instance of a class is called an *object*.

Check your knowledge

1. Which of the following represents the correct way to create a class instance? *

- ◯ `Random dice = new Random.Next();`.
- ◯ `Random dice = new Random();`.
- ◯ `String dice = new Random();`.

2. A developer creates an instance of the `Random` class named `coins`. Which of the following code lines can they use to call the `Next()` method? *

- ◯ `int money = new coins.Next();`.
- ◯ `int money = Random.Next();`.
- ◯ `int money = coins.Next();`.

Check your answers

Module complete:

Unlock achievement