200 XP

Exercise - Return values and input parameters of methods

18 minutes

In the previous unit, you used a "roll dice" coding scenario to illustrate the difference between stateful (instance) and stateless (static) methods. That same scenario can help you to understand other important concepts about calling methods. For example:

- handling the return value of a method.
- passing input parameters to a method.
- choosing an overloaded version of a method.

Return values

Some methods are designed to complete their function and end "quietly". In other words, they don't return a value when they finish. They are referred to as **void methods**.

Other methods are designed to return a value upon completion. The return value is typically the result of an operation. A return value is the primary way for a method to communicate back to the code that calls the method.

You saw that the `Random.Next()` method returns an `int` type containing the value of the randomly generated number. However, a method can be designed to return any data type, even another class. For example, the `String` class has some methods that return a string, some that return an integer, and some that return a Boolean.

When calling a method that returns a value, you'll often assign the return value to a variable. That way, you can use the value later in your code. In the dice scenario, you assigned the return value of `Random.Next()` to the `roll` variable:

C#

```
int roll = dice.Next(1, 7);
```

In some cases, you might want to use the return value directly, without assigning it to a variable. For example, you might want to print the return value to the console as follows:

```c#
Console.WriteLine(dice.Next(1, 7));
```

Even though a method returns a value, it's possible to call the method without using the return value. For example, you could ignore the return value by calling the method as follows:

```c#
dice.Next(1, 7);
```

However, ignoring the return value would be pointless. The reason you're calling the `Next()` method is so that you can retrieve the next random value.

Input parameters

The information consumed by a method is called a parameter. A method can use one or more parameters to accomplish its task, or none at all.

> **Note**
>
> Often times, the terms 'parameter' and 'argument' are used interchangeably. However, 'parameter' refers to the variable that's being used inside the method. An 'argument' is the value that's passed when the method is called.

Most methods are designed to accept one or more input parameters. The input parameters can be used to configure how the method performs its work, or they might be operated on directly. For example, the `Random.Next()` method uses input parameters to configure the upper and lower boundaries of the return value. However, the `Console.WriteLine()` uses the input parameter directly by printing the value to the console.

Methods use a **method signature** to define the number of input parameters that the method will accept, as well as the data type of each parameter. The coding statement that calls the method must adhere the requirements specified by the method signature. Some methods provide options for the number and type of parameters that the method accepts.

When a caller invokes the method, it provides concrete values, called arguments, for each parameter. The arguments must be compatible with the parameter type. However, 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.

Consider the following code:

```
C#

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

The first code line creates an instance of the `Random` class named `dice`. The second code line uses the `dice.Next(1, 7)` method to assign a random value to an integer named `roll`. Notice that the calling statement provides two arguments separated by a `,` symbol. The `Next()` method includes a method signature that accepts two input parameters of type `int`. These parameters are used to configure the lower and upper boundaries for the random number that's returned. The final code line uses the `Console.WriteLine()` method to print the value of `roll` to the console.

The arguments passed to a method must be the same data type as the corresponding input parameters defined by the method. If you attempt to pass an incorrectly typed argument to a method, the C# compiler will catch your mistake and force you to update your calling statement before your code will compile and run. Type checking is one way that C# and .NET use to prevent end-users from experiencing errors at runtime.

> ⓘ **Note**
>
> Although input parameters are often used, not all methods require input parameters to complete their task. For example, the `Console` class includes a `Console.Clear()` method that doesn't use input parameters. Since this method is used to clear any information displayed in the console, it doesn't need input parameters to complete it's task.

Overloaded methods

Many methods in the .NET Class Library have *overloaded* method signatures. Among other things, this enables you to call the method with or without arguments specified in the calling statement.

An **overloaded method** is defined with multiple method signatures. Overloaded methods provide different ways to call the method or provide different types of data.

In some cases, overloaded versions of a method are used to define an input parameter using different data types. For example, the `Console.WriteLine()` method has 19 different overloaded versions. Most of those overloads allow the method to accept different types and then write the specified information to the console. Consider the following code:

```
c#

int number = 7;
string text = "seven";

Console.WriteLine(number);
Console.WriteLine();
Console.WriteLine(text);
```

In this example, you're invoking three separate overloaded versions of the `WriteLine()` method.

- The first `WriteLine()` method uses a method signature that defines an `int` parameter.
- The second `WriteLine()` method uses a method signature that defines zero input parameters.
- The third `WriteLine()` method uses a method signature that defines a `string` parameter.

In other cases, overloaded versions of a method define a different number of input parameters. The alternative input parameters can be used to provide more control over desired result. For example, the `Random.Next()` method has overloaded versions that enable you to set various levels of constraint on the randomly generated number.

The following exercise calls the `Random.Next()` method to generate random integer values with different levels of constraint:

1. Ensure that you have an empty Program.cs file open in Visual Studio Code.

   If necessary, open Visual Studio Code, and then complete the following steps to prepare a Program.cs file in the Editor:

   a. On the **File** menu, select **Open Folder**.

   b. Use the Open Folder dialog to navigate to, and then open, the **CsharpProjects** folder.

   c. In the Visual Studio Code EXPLORER panel, select **Program.cs**.

d. On the Visual Studio Code **Selection** menu, select **Select All**, and then press the Delete key.

2. To examine the overloaded versions of the `Random.Next()` method, enter the following code:

```
c#

Random dice = new Random();
int roll1 = dice.Next();
int roll2 = dice.Next(101);
int roll3 = dice.Next(50, 101);

Console.WriteLine($"First roll: {roll1}");
Console.WriteLine($"Second roll: {roll2}");
Console.WriteLine($"Third roll: {roll3}");
```

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

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

   Ensure that folder path displayed in the command prompt points to the folder containing your Program.cs file.

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

   Notice that your result is similar to the following output:

```
Output

First roll: 342585470
Second roll: 43
Third roll: 89
```

   The numbers generated are random, so your results will be different. However, this example demonstrates the range of results that you might see.

6. Take a minute to examine the code.

   The first version of the `Next()` method doesn't set an upper and lower boundary, so the method will return values ranging from `0` to `2,147,483,647`, which is the maximum value an `int` can store.

The second version of the `Next()` method specifies the maximum value as an upper boundary, so in this case, you can expect a random value between `0` and `100`.

The third version of the `Next()` method specifies both the minimum and maximum values, so in this case, you can expect a random value between `50` and `100`.

7. Close the Terminal panel.

You've already examined several topics in this unit. Here's a quick list of what you've covered:

- You've examined how to use a method's return value (when the method provides a return value).
- You've examined how a method can use input parameters that are defined as specific data types.
- You've examined the overloaded versions of some methods that include different input parameters or parameter types.

Use IntelliSense

Visual Studio Code includes *IntelliSense* features that are powered by a language service. For example, the C# language service provides intelligent code completions based on language semantics and an analysis of your source code. In this section, you'll use IntelliSense to help you implement the `Random.Next()` method.

Since IntelliSense is exposed within the code editor, you can learn a lot about a method without leaving the coding environment. IntelliSense provides hints and reference information in a popup window under the cursor location as you enter your code. When you are typing code, the IntelliSense popup window will change its contents depending on the context.

For example, as you enter the word `dice` slowly, IntelliSense will show all C# keywords, identifiers (or rather, variable names in the code), and classes in the .NET Class Library that match the letters being entered. Autocomplete features of the code editor can be used to finish typing the word that is the top match in the IntelliSense popup. Try it out.

1. Ensure that you have your Program.cs file open in Visual Studio Code.

   Your app should contain the following code:

   ```
   c#
   ```

```
Random dice = new Random();
int roll1 = dice.Next();
int roll2 = dice.Next(101);
int roll3 = dice.Next(50, 101);

Console.WriteLine($"First roll: {roll1}");
Console.WriteLine($"Second roll: {roll2}");
Console.WriteLine($"Third roll: {roll3}");
```

2. At the bottom of your code file, to experiment with IntelliSense, slowly enter the letters `d`, `i` then `c`.

3. Notice the IntelliSense popup window that appears when you begin typing.

   When IntelliSense pops up, a list of suggestions should appear. By the time you have entered `dic`, the identifier `dice` should be at the top of the list.

4. Press the Tab key on the keyboard.

   Notice that the entire word `dice` is completed in the editor. You can use the up and down arrow keys to change the selection before pressing the Tab key.

   > **Note**
   >
   > If the IntelliSense window disappears, it can be selected by using the `backspace` key on the keyboard, then re-enter the last symbol to re-open IntelliSense.

5. To specify the member access operator, enter a `.` character.

   Notice that the IntelliSense popup reappears when you enter `.` and shows an unfiltered list of all the methods (and other members of the class) that are available.

6. Enter **N**

   The list will be filtered, and the word `Next` should be the top selection.

7. To autocomplete the entire word, press the Tab key.

8. To specify the method invocation operator, enter **(**

   Notice that the closing parenthesis is automatically added for you.

The method invocation operator is the set of parentheses located to the right of the method name. This portion of the calling statement is where you specify the arguments that will be passed to the method. The method invocation operator is required when calling the method.

9. Notice that the IntelliSense popup now displays detailed information about the `Random.Next()` method.

10. Take a minute to examine the IntelliSense popup for the `Random.Next()` method.

> **Note**
>
> If the IntelliSense popup closed before you had a chance to examine it, delete the invocation operator `()`, and then enter `(` to display the IntelliSense popup.

Notice that the popup window includes three sections, one on the left and two on the right.

On the right side, you should see `int Random.Next(int minValue, int maxValue)` in the top section, and `Returns a non-negative random integer.` in the bottom section. The `int` defines the return type for the method. In other words, when this version of the method is executed, it will return a value of type `int`.

On the left side of the IntelliSense popup, it displays `1/3`.

The `1/3` indicates that you're looking at the first of three method signatures for the `Next()` method. Notice that this version of the method signature enables the method to work with no input parameters (no arguments passed to the method in the calling statement).

Notice that there's also a tiny arrow above and below the `1/3`.

11. To examine the second overloaded version of the method, press the Down Arrow key on the keyboard.

Notice that you can use the up and down arrow keys to navigate between the various overloaded versions. When you do, you'll see the `1/3`, `2/3`, and `3/3` appear on the left side of the IntelliSense popup, and helpful explanations on the right.

12. Take a minute to examine each of the overloaded versions for the `Random.Next()` method.

The second overloaded version of the method, `2/3`, informs you that the `Next()` method can accept an input parameter `int maxValue`. The description tells you that `maxValue` is the exclusive upper bound for the number that you want the `Next()` method to generate. Exclusive indicates that the return number will be less than maxValue. So when you specify `dice.Next(1,7);` the max dice roll will be 6. Notice that the message at the bottom of the section has been updated to: `Returns a non-negative random integer that is less than the specified maximum.`

The third version of the method, `3/3`, informs you that the `Next()` method can accept both `int minValue` and `int maxValue` as input parameters. The new parameter, `minValue`, is a lower bound for the number that you want the `Next()` method to generate. Since the lower bound is inclusive rather than exclusive, the return value can be equal to `minValue`. The message at the bottom now states: `Returns a random integer that is within a specified range.`

In this case, IntelliSense provides all of the information that you need to select the appropriate overload, including a detailed explanation of `maxValue` and `minValue`. However, you might encounter situations where you need to consult the method's documentation.

Use learn.microsoft.com for information about overloaded methods

The second way to learn about overloaded versions of the methods is to consult the documentation for the method. The documentation will also help you to understand exactly what each input parameter is intended for.

1. To begin, open your preferred Web browser and search engine.
2. Perform a search for **C# Random.Next()**

   Your search should include the class name and method name. You might also want to include the term `C#` to make sure not to accidentally get results for other programming languages.
3. Select the top search result with a URL that begins with `https://learn.microsoft.com`.

   One of the top search results should lead to a URL that begins with `https://learn.microsoft.com`. In this case, the link's title should appear as `Random.Next`

`Method`.

Here's the link in case you have a problem finding it using a search engine:

Random.Next Method

4. Open the link for **C# Random.Next()**.

5. Quickly scan through the documentation.

   Scroll down through the page contents to see the various code samples. Notice that you can run the samples in the browser window.

   The learn.microsoft.com documentation follows a standard format for each class and method in the .NET Class Library.

6. Near the top of the web page, locate the section labeled **Overloads**.

   Notice that there are three overloaded versions of the method listed. Each overloaded version that's listed includes a hyperlink to a location further down on the page.

7. To navigate "on-page" to a description of the second overloaded version, select **Next(Int32)**.

   Documentation for each version of the method includes:

   - Brief description of the method's functionality
   - Method's definition
   - Input parameters that the method accepts
   - Return values
   - Exceptions that can be raised
   - Examples of the method in use
   - Other remarks about the method

8. Take a minute to review the **Parameters** section.

   In the *Parameters* section, you can read that the `maxValue` input parameter is the "exclusive upper bound of the random number to be generated." An *exclusive upper bound* means that if you want numbers no larger than `10`, you must pass in the value `11`.

   You can also read in the next line: "`maxValue` must be greater than or equal to 0." What happens if you ignore this statement? You can see in the *Exceptions* section that the method will return an `ArgumentOutOfRangeException` when `maxValue` is less than 0.

> **Note**
>
> The content at learn.microsoft.com is the "source of truth" for the .NET Class Library. It's important to take the time to read the documentation to understand how a given method will work.

Recap

- Methods might accept no parameters or multiple parameters, depending on how they were designed and implemented. When passing in multiple input parameters, separate them with a `,` symbol.
- Methods might return a value when they complete their task, or they might return nothing (void).
- Overloaded methods support several implementations of the method, each with a unique method signature (the number of input parameters and the data type of each input parameter).
- IntelliSense can help write code more quickly. It provides a quick reference to methods, their return values, their overloaded versions, and the types of their input parameters.
- learn.microsoft.com is the "source of truth" when you want to learn how methods in the .NET Class Library work.

Check your knowledge

1. What is a return value? *

- ◯ It's a value type returned by a method.
- ◯ An argument in a method call is referred to as a return value inside the method.
- ◯ It's a string value.

2. What are input parameters? *

- ◯ Value types (or variables) inside a method.
- ◯ Values returned by a method.

- ◯ The values passed into a method in the calling statement.

3. What is an overloaded method? *

- ◯ A method that returns a value type.
- ◯ A method with more than five parameters.
- ◯ It is a method that supports several implementations of the method, each with a unique method signature.

4. How does IntelliSense help developers? *

- ◯ IntelliSense can help developers write code more quickly.
- ◯ IntelliSense helps developers refactor their code.
- ◯ IntelliSense can change the "theme" of an IDE.

Check your answers

Module complete:

Unlock achievement