

Module 5 – Methods in C#.

When developing a software, you will often need to perform a specific operation (e.g., printing a text in a specific format) in several parts of your code. Performing this operation might cost you 50 lines of code. If **methods** never existed, you would need to rewrite these 50 lines to perform the same operation, which would create a lot of redundancy in your code. What if you have to change something about this operation? It could take hours of effort each time to change your statements properly and this process would possibly result in new bugs. In other words, your program would become difficult to maintain. So, this is the first reason why we need methods: to minimize redundancy and to maximize maintainability.

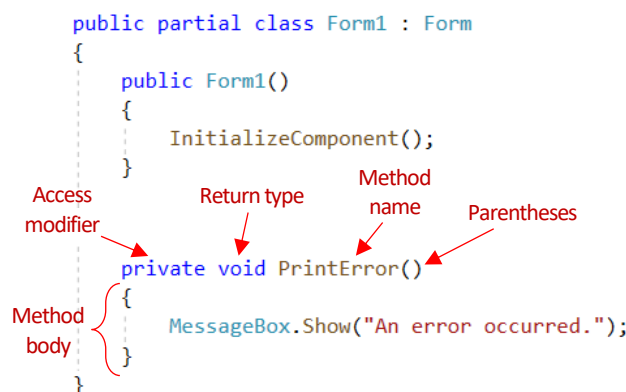
Second, complex programming tasks may require you to write hundreds of lines of code. It is obviously difficult for a programmer to structure such a large chunk of code and apply the intended programming logic easily. Such complex tasks can be dealt with rather easily if divided into smaller subtasks. Separate methods can be defined to perform these distinct (but related) subtasks, and then can be executed in the order necessary to perform the complex task. Thus, methods can also help us modularize our code by dividing it into manageable (and rather smaller) chunks.

Thus far, we have used some methods already. The examples include `ToString`, `MessageBox.Show`, and `TryParse`. In this chapter we will learn how to create our own custom methods. Before getting started with writing code, please create a new project named `Module6`.

1) Defining a method

A method definition in C# (as in many other languages) consists of a header and a body. The header helps you configure the method, and the body contains the statements to be executed when the method is called. In the following example (see Figure 1), a method called `PrintError` is defined inside the `Form1` class (just under the `public Form1()` code block). In C#, methods are very often defined inside a class.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
    private void PrintError()
    {
        MessageBox.Show("An error occurred.");
    }
}
```



The diagram shows the `PrintError` method definition within the `Form1` class. Red arrows point to the following parts of the code:
- **Access modifier**: points to `private`.
- **Return type**: points to `void`.
- **Method name**: points to `PrintError`.
- **Parentheses**: points to the empty parentheses `()`.
- **Method body**: indicated by a bracket on the left side of the curly braces `{ ... }` surrounding the `MessageBox.Show` statement.

Figure 1. Defining a method called `PrintError`.

The header of a method is composed of four parts as shown in Figure 1. *Access modifier* determines if the method can be called outside of the class (i.e., `public`) or it is only accessible inside the same class (i.e.,

`private`). *Return type* determines the data type of the value that is returned by the method if any. If no value is returned, then `void` is used as the data type.

Method name is used when calling the method and it should be descriptive. The rules that apply to variable names are also valid for method names. It is recommended to use **PascalCase** in method names, which is quite similar to camelCase that we have been using when defining variable names except the first letter should be also uppercase. *Parentheses* are used optionally to indicate some parameters to pass some values inside the method. Method body contains the statements between braces `{ }` that are executed linearly when the method is called.

2) Calling a method

Once we have the method defined, we have to make a call to the method to execute it. We will add a button to the form as shown in Figure 2. Clicking on this button will execute the **PrintError** method.

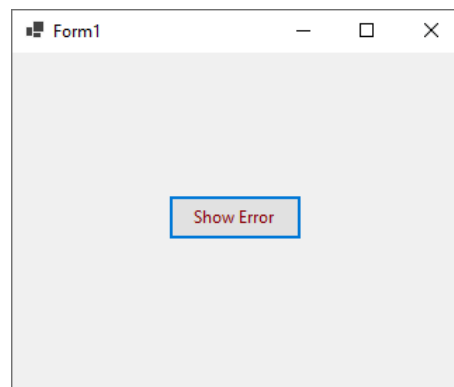


Figure 2. Adding a button to the form.

Double click on the button to create its click event handler. Actually, click event handler is a method that is executed only when the associated button is clicked. The components of the click event handler as a method are shown in Figure 3.

```
private void btn_showError_Click(object sender, EventArgs e)
{
}
```

Access modifier Return type Method name Parameters inside the parentheses

Figure 3. Click event handler as a method.

Now we will make a call to the method inside the click event handler. To do so, you just need to write the method name and add parenthesis at the end. Since, the method does not accept parameters, inside the parenthesis will be empty. Before and after the method call, we will print “Hello” and “End” messages. The complete code is provided in Figure 4.

```

private void PrintError()
{
    MessageBox.Show("An error occurred.");
}

private void btn_showError_Click(object sender, EventArgs e)
{
    MessageBox.Show("Hello");
    PrintError();
    MessageBox.Show("End");
}

```

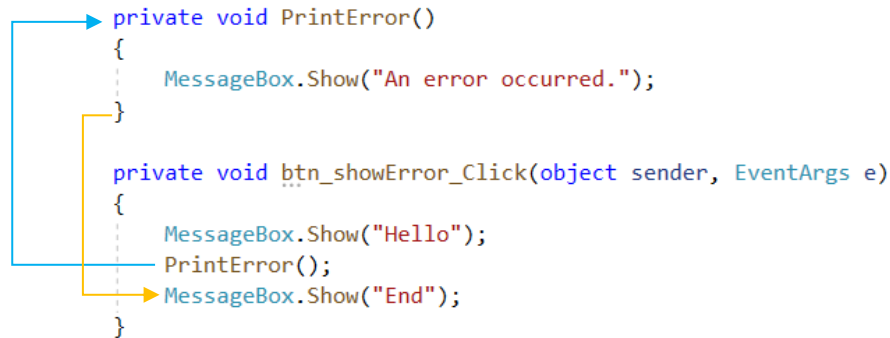


Figure 4. Click event handler code.

As illustrated in Figure 4, when the program executes the method call, it will jump to the method definition (wherever it is defined inside the class). After the execution of the method is completed, the program will jump to the next line after the method call.

Please run your application by pressing Control+F5. Clicking on the button, will display first the Hello message, then the Error text, and last the End message, as shown in Figure 5, 6, and 7.

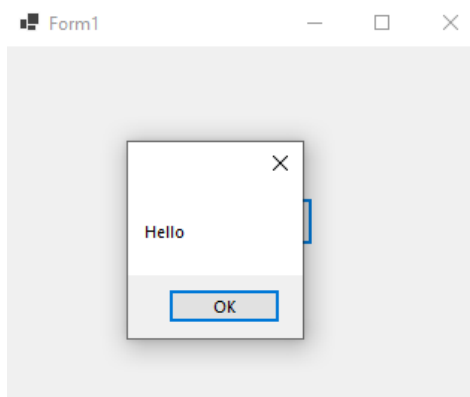


Figure 5. Hello message displayed.

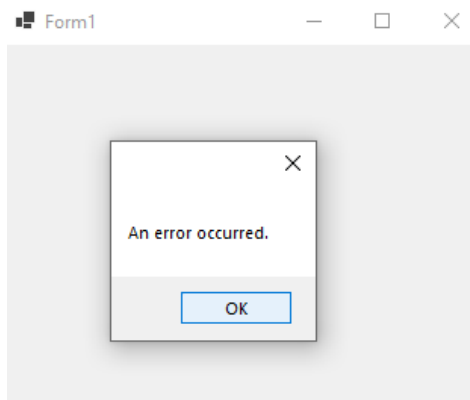


Figure 6. Error message displayed.

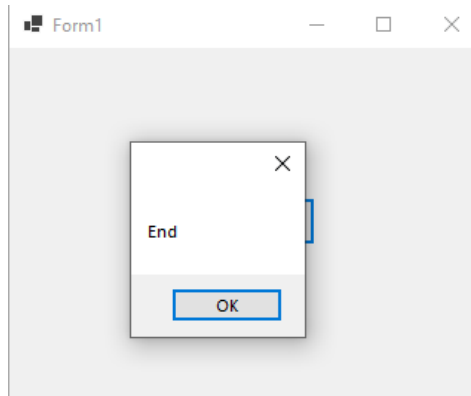


Figure 7. End message displayed.

3) Passing an argument

Now, we will change our example a bit. We will create an application in which we will allow a user to increment or decrement the number entered inside a textbox. Delete the button added previously. Please create the following form design (see Figure 8).

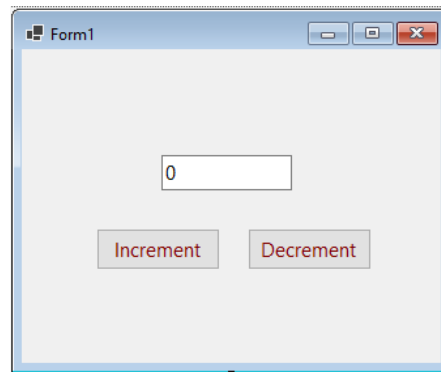


Figure 8. The design of the form.

Next, switch to the Code view (by pressing F7). Delete the previous method definitions and create the `Increment` and `Decrement` methods as shown in Figure 9. These methods will require an `int` parameter as indicated inside the parentheses in their headers. In other words, we will HAVE TO pass some **arguments** to be used by the method in its body.

```
private void Increment(int number)
{
    number = number + 1; //number++
    txt_number.Text = number.ToString();
}

private void Decrement(int number)
{
    number = number - 1; //number--
    txt_number.Text = number.ToString();
}
```

Figure 9. Defining the Increment and Decrement methods.

While the `Increment` method increases the value of the number by 1 and assigns the incremented value back to the textbox, the `Decrement` method decreases the value of the number by 1. We will call these methods from the click event handlers of the buttons we created.

Switch to the design view (Control+F7) and double click on the Increment button to create its click event handler. Then, add the code following code shown in Figure 10 inside the handler.

```
if(int.TryParse(txt_number.Text, out int myNumber))
{
    Increment(myNumber);
}
else
{
    MessageBox.Show("Invalid entry.");
}
```

Figure 10. Click event handler for the Increment button.

As you may remember, clicking on this button should read the value entered inside the textbox, increase its value by 1, and then print it inside the textbox. To achieve this, we first use the `TryParse` method to check if the textbox has a valid `int` entry. If so, the `TryParse` method will convert what is entered inside the textbox into `int` type and store the converted value inside the variable called `myNumber`. Then, we call the `Increment` method and pass the `myNumber` variable as the argument. As we defined earlier, the `Increment` method should increase the value of `myNumber` by 1 and show it inside the textbox.

We will almost have the same code for the click event handler of the Decrement button except that instead of calling the `Increment` method, we will call the `Decrement` method. See Figure 11 for the details.

```
private void btn_decrement_Click(object sender, EventArgs e)
{
    if (int.TryParse(txt_number.Text, out int myNumber))
    {
        Decrement(myNumber);
    }
    else
    {
        MessageBox.Show("Invalid entry.");
    }
}
```

Figure 11. Click event handler for the Decrement button.

Press Control+F5 to test the application. Enter 40 inside the textbox as shown in Figure 12 and click the Increment button only once. The value 40 will change to 41 (see Figure 13). If you click more, the value will increment one by one. Now, click twice on the Decrement button. The value should change to first 40, and then 39 (see Figure 14).

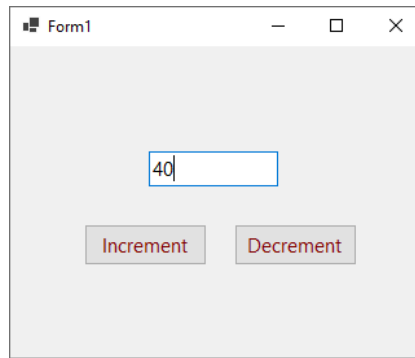


Figure 12. Entering 40 inside the textbox.

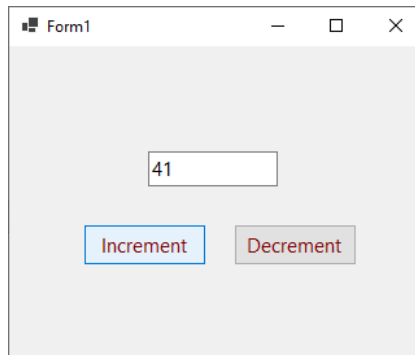


Figure 13. Clicking the Increment button only once.

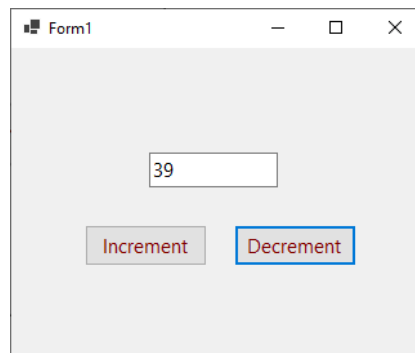


Figure 14. Clicking the Decrement button twice.

4) Returning a value from a method

The methods we defined so far are **void** methods that do not return any value. However, we can define a method that can return a value. We will change the void `Decrement` and `Increment` methods and convert them into **value-returning** methods.

We need to first determine what data type the method will return. For example, the `Increment` method increases the value of the number (passed as argument). We want the method to return this updated value, which is an integer type, instead of printing it in the textbox. Thus, the return type of the method should be set to `int` in its header.

The value to be returned must be preceded by the **return** keyword. Without a `return` statement, you would get an error since the method is not defined as **void**. With the `return` statement, the method execution terminates, and the value is sent to where the call to the method is made.

The definition of `Increment` as a value-returning method is provided in Figure 15. Note that its return type is set to **int** (inside the method header), and a `return` statement is added inside the method body.

```
private int Increment(int number)
{
    number = number + 1; //number++
    return number;
}
```

Figure 15. Increment as a value-returning method.

Similarly, please update the `Decrement` method as shown in Figure 16.

```
private int Decrement(int number)
{
    number = number - 1; //number--
    return number;
}
```

Figure 16. Decrement as a value-returning method.

We need to do some changes in the statement where we make a call to these methods since now they return the updated number (instead of printing it).

The returned value can be used like any other value. It can be assigned to a variable (`int newValue = Increment(number)`) or displayed on the screen. We will assign the return value to the **Text** property of the textbox to display the updated value. See Figure 17 for the complete code for both methods.

```
private void btn_increment_Click(object sender, EventArgs e)
{
    if (int.TryParse(txt_number.Text, out int myNumber))
    {
        txt_number.Text = Increment(myNumber).ToString();
    }
    else
    {
        MessageBox.Show("Invalid entry.");
    }
}

private void btn_decrement_Click(object sender, EventArgs e)
{
    if (int.TryParse(txt_number.Text, out int myNumber))
    {
        txt_number.Text = Decrement(myNumber).ToString();
    }
    else
    {
        MessageBox.Show("Invalid entry.");
    }
}
```

Figure 17. Using the returned value from the methods.

You can run and test the application. It should work the same way.

7) Passing reference parameters

In the example so far, the arguments passed inside the methods are the copies of the values. For example, calling the `Decrement(myNumber)` method makes a local copy of `myNumber` and perform these operations on the local copy not on the `myNumber` variable itself. That is, the operations performed inside the method will not affect the value of `myNumber`.

We can use the `ref` keyword to create reference type parameters. When `ref` keyword is provided before the argument name, a reference to where the argument is stored in the memory is created, and this reference value is passed, not the copy of the argument value. In this case, any possible changes made on the reference type parameters will affect the value of the actual variable.

We will update the `Increment` method to use the reference type parameters. The changes on the existing method are shown in the following code. First, we will replace `int` with `void` in the header of the method, since now our method will not return any value. Next, we will add the `ref` keyword before the definition of the parameter named `number`. Last, we will remove the `return` statement.

```
private void int Increment(ref int number)
{
    number = number + 1;
    return number;
}
```

Since we changed the method definition, we also need to change the code where we make a call to the method to use it, which is inside the button click event. We will make a call to the method without assigning to a variable or to the Text property of the textbox (as we did previously). Since the method changes the value of `myNumber` variable, after calling the method, `myNumber` will increment by one. In the subsequent statement, we will directly assign the `myNumber` variable to the text property of the textbox. Your code should look like this:

```
private void btn_increment_Click(object sender, EventArgs e)
{
    if (int.TryParse(txt_number.Text, out int myNumber))
    {
        Increment(ref myNumber); //myNumber is incremented
        txt_number.Text = myNumber.ToString(); Increment(myNumber).ToString();
    }
    else
    {
        MessageBox.Show("Invalid entry.");
    }
}
```


8) Using out keyword

Sometimes we need a method to return two separate values (of any data type) and use these values for different purposes. **TryParse**, that we have been using for a while, is actually a great example for this. For example, `int.TryParse` returns a Boolean value (true or false) to indicate if the conversion to int is successful or not, and also it returns the converted value. The converted value is stored inside the argument that was passed with the **out** keyword.

We will create our own custom method that takes advantage of the **out** keyword. Before diving into the code, we will add a new functionality into our application: **division**. As shown in Figure 18, please add a textbox (named `txt_denominator`) and a button (named `btn_divide`) at the bottom part of the form.

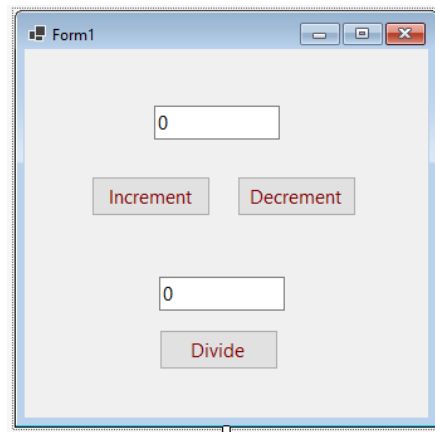


Figure 18. New controls for division.

We will create a new method called **Divide**, which will return a Boolean value if the division is successful or not, as well as the resulting value after the division. The return type of this method will be `bool`. It will return `false` if the division fails and `true` if the division succeeds. The method will accept three parameters: a parameter to pass the `nominator` value, a parameter to pass the `denominator` value, and an **out** parameter, named `result`, to store the result of the division operation. Below (see Figure 19) is the definition of the `Divide` method.

```
private bool Divide(int nominator, int denominator, out int result)
{
}
}
```

Figure 19. Header of the Divide method.

Inside the method, we can perform the division using the `/` operator, and assign the output to the `result` variable and return `true` as shown in Figure 20.

```
private bool Divide(int nominator, int denominator, out int result)
{
    result = nominator / denominator;
    return true;
}
```

Figure 20. Computing the division and returning true.

However, this code will not be sufficient if the denominator is 0. Since dividing an integer by 0 will throw **DivideByZeroException** and the program will halt. To prevent this, we will use `try-catch`. We will insert the existing code inside the `try` block, and we will return `false` inside the `catch` block and set the value of `result` to 0. Please note that before any return statement, the `out` variable has to be set a value. Otherwise, you will receive an error. Below is the complete definition of the `Divide` method.

```
private bool Divide(int nominator, int denominator, out int result)
{
    try
    {
        result = nominator / denominator;
        return true;
    }
    catch
    {
        result = 0;
        return false;
    }
}
```

Figure 21. Complete definition of the `Divide` method.

After defining the `Divide` method, please switch back to the design view and double click on the `Divide` button to create its click event handler. Inside this handler, we will call the `Divide` method. We will pass three arguments: besides the `nominator` and `denominator` values, we need to pass an **out** variable in which the division result will be stored. Since our method returns a `bool` type, we will assign it directly to a `bool` variable called `isDivisionSuccessful`. The details are provided below.

```
private void btn_divide_Click(object sender, EventArgs e)
{
    //return value, which is true or false, is assigned to a new variable
    bool isDivisionSuccessful = Divide(nominator, denominator, out int result);
    //the result of the division will be stored in the result parameter
}
```

Figure 22. Calling the `Divide` method.

Please note that `nominator` and `denominator` variables will be defined later. After calling the method, by checking the value of `isDivisionSuccessful`, we can decide whether to update the textbox with the division output or to print the error message about the failure of the division. For this purpose, we will use `if-else` statement as shown below.

```
private void btn_divide_Click(object sender, EventArgs e)
{
    //return value, which is true or false, is assigned to a new variable
    bool isDivisionSuccessful = Divide(nominator, denominator, out int result);
    //the result of the division will be stored in the result parameter

    if (isDivisionSuccessful)//display the result if the division succeeds.
    {
        txt_number.Text = result.ToString();
    }
    else//if the division fails, display an error message.
    {
        MessageBox.Show("Division failed.");
    }
}
```

Figure 23. Calling the `Divide` method.

Before continuing with the rest of the code, we will create a **region**. Regions can help you expand or collapse parts of your code. You can use regions to hide a part of your code that is complete and switch your attention to another part. We will create a new region for the code we have for now inside the click event handler. While **#region** starts a region, **#endregion** ends the region started. Figure 24 shows the code to create a region called COMPUTE DIVISION. Once the division is created, the small squares on the left-hand side to hide (or show) the region will appear.

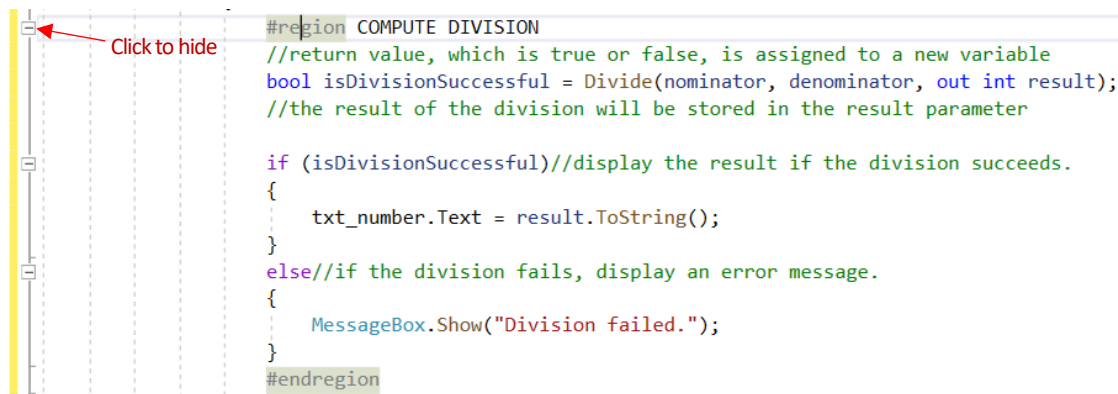


Figure 24. Creating a region.

Please click to hide the region as shown in Figure 24. Now, we will read the input from the `txt_denominator` textbox and store it inside a new variable called `denominator`. For this purpose, we will use **TryParse** method as shown in Figure 25 below. The **out** keyword will accompany the `denominator` parameter, which will hold the converted value. The *COMPUTE DIVISION* region will be included inside the `if` block. That means, the code inside this region will be executed only if the **TryParse** method returns true (i.e., users enter a valid number in the textbox).

```
private void btn_divide_Click(object sender, EventArgs e)
{
    if (int.TryParse(txt_denominator.Text, out int denominator))
    {
        COMPUTE DIVISION
    }
    else
    {
        MessageBox.Show("Invalid denominator value.");
    }
}
```

Figure 25. Checking for a valid denominator value.

If the value returned from the `TryParse` method is false (i.e., user enters an invalid number), then an error message will be displayed.

We will repeat the same logic to ensure that the value entered inside the `txt_number` textbox is a valid numeric value. We will create a new `if-else` statement and inside its `if` block we will place our existing code. That is, we will create a **nested** `if-else` decision structure. The complete code is shown in Figure 26.

```

private void btn_divide_Click(object sender, EventArgs e)
{
    if (int.TryParse(txt_number.Text, out int nominator))
    {
        if (int.TryParse(txt_denominator.Text, out int denominator))
        {
            //THIS CODE BLOCK IS ACCESSED ONLY IF BOTH nominator & denominator ARE VALID.
            COMPUTE DIVISION
        }
        else
        {
            MessageBox.Show("Invalid denominator value.");
        }
    }
    else
    {
        MessageBox.Show("Invalid nominator entry.");
    }
}

```

Figure 26. Checking for a valid nominator value.

Now you can test the application. It should display proper error messages with invalid user inputs and should never crash.

As a further exercise, you can define a single more generic method instead of Decrement and Increment method. This generic method should accept a new parameter to indicate the type of the operation (increment vs decrement). This will be covered in the class if the time allows.