

Module 4 – Variables and numeric user inputs.

In the last module, we have started to get our hands dirty with C# coding. This week, we will build on that by learning how to assign user input to variables and how to handle numeric user inputs

Before moving forward, please create new empty C# windows forms project in Visual Studio that we will use to implement the examples during this chapter. You can give any name to the project (and solution). In our example, it will be *Module4_1*.

1) Using variables

One of the basic concepts in programming is the variables. A variable refers to a location in the memory (of the computer) that holds a value. You may remember from the application we developed in the previous module (see Figure 1) that we did not use any variables to store the first name or last name values and, instead, directly use this values to produce the full name (see Figure 2).

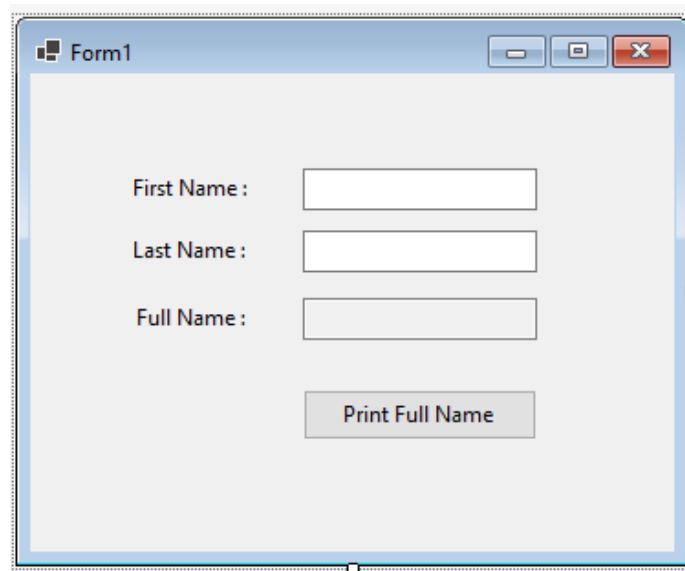


Figure 1. Form design of the application to print full name.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private void btn1_printFullName_Click(object sender, EventArgs e)
    {
        txt_fullName.Text = txt_firstName.Text + " " + txt_lastName.Text;
    }
}
```

Figure 2. Complete code for printing full name.

However, it is often more advantageous to hold these values in some variables since in different parts of the code we may need to refer to these variable depending on our programming logic. Using variables can also increase the readability of our code. Since variables occupies the memory, they should be used wisely. Otherwise, our application may occupy too much memory and lead to performance issues.

Before using a variable, we first need to define (or declare) it. In C#, this is done using the following syntax: `dataType variableName;`. We choose data type depending on what kind of data we want to store. For example, if you want to store a number, the data type might be `int`, or if you want to store a text, the data type would be `string`.

Right now, we will focus on `string` variables. `string` variables hold any textual data (names, addresses, passwords, etc.). We will do slight changes in our code and enhance it by creating several `string` variables. First, we will define a string variable called `firstName` to store the value typed into `txt_firstName`. Next, we will define another string variable (`lastName`) to store the value of the `txt_lastName`. Figure 3 provides the code to create these variables and to store the user input. Please note that the variable names should be concise and explanatory. For example, instead of `firstName` we could use `x` as the variable name, but it would not communicate the purpose of the variable.

```
private void btn1_printFullName_Click(object sender, EventArgs e)
{
    string firstName; //Define the variable
    firstName = txt_firstName.Text; //Store the user's first name

    //Define the variable and store the user's last name
    string lastName = txt_lastName.Text;
}
```

Figure 3. Defining the variables to store user input.

As you may notice, we have followed two different approaches. In the first approach, the `firstName` variable was created, and then in the following line, the value stored in the **Text** property of the `txt_firstName` is assigned to this variable. In the second approach, the variable creation (`lastName`) and the value assignment (`txt_lastName.Text`) took place in the same line. The variable names are automatically coloured with light grey, to warn us that they have not been used yet. Unused variables will occupy unnecessary space in the memory and hurt the performance of your application.

We will use these variables to produce the full name and store the full name inside a new variable called `fullName`. Then, we will assign this variable to the Text property of `txt_fullName` to print the `fullName` in the form. See the complete code provided in Figure 4.

```

private void btn1_printFullName_Click(object sender, EventArgs e)
{
    string firstName;//Define the variable
    firstName = txt_firstName.Text;//Store the user's first name

    //Define the variable and store the user's last name
    string lastName = txt_lastName.Text;

    string fullName = firstName + " " + lastName;//Produce the fullName and store it.
    txt_fullName.Text = fullName;//Print the fullname inside the textbox
}

```

Figure 4. The complete code to print full name.

You can run and test the application. It should work the same way as before. However, with variables we have a better structured and explained code than before.

2) Scope of variables

Accessibility of variables depends on their scope. For example, in the example above (see Figure 4), all variables are local variables since they were created inside the click event handler method. They can be considered to have **method level scope**. Therefore, they are ONLY accessible within this method, and outside they are undefined. These variables will be deleted from the memory once the execution of this method is completed.

To test the accessibility of the variables, we will add a new button, which will print full name in a reversed manner (Last Name, First Name). Change its **Name** and the **Text** properties as suggested in Figure 5.

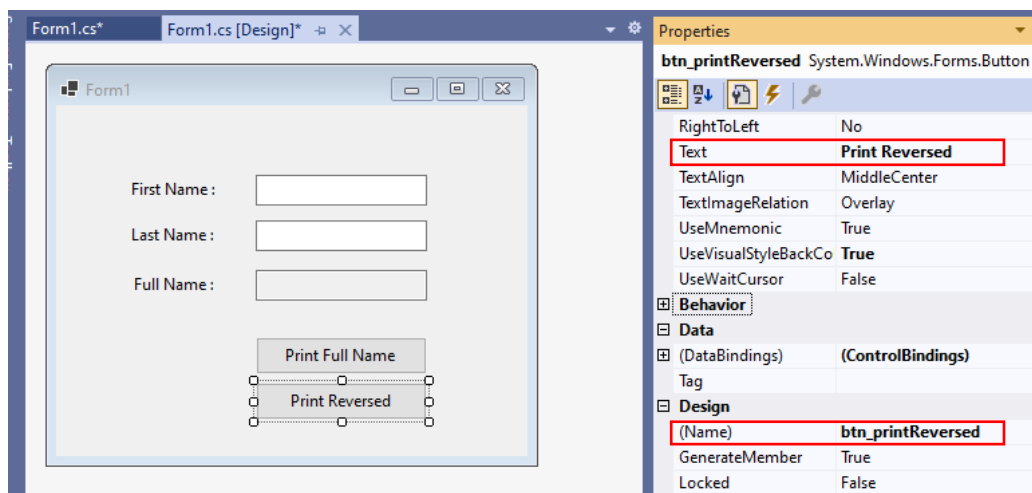


Figure 5. Adding a new button to print reverse full name.

Then, double click on `btn_printReversed` to create its click event handler as shown in Figure 6.

```

private void btn_printReversed_Click(object sender, EventArgs e)
{
}

```

Figure 6. Click event handler for `btn_printReversed`.

Print Reversed button will almost do the same thing as the *Print Full Name* button, except that it will first print the last name, and then the last name. So, we might want to use the same variables that we created for the click event handler of the *Print Full Name* button. As in Figure 7, if you begin to type initial letters of the `firstName` variable, it will not appear in the popup list suggesting matching items. In the list, the only matching item is `txt_firstName`, which is not what we want.

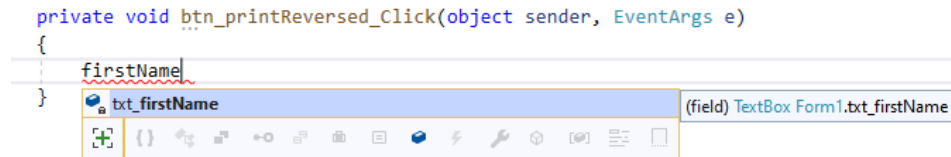


Figure 7. Accessing `firstName` inside the `btn_printReversed_Click` event handler.

This is because we are now in a different scope than where the variable `firstName` was defined. This variable was defined inside the `btn_printFullName` event handler, and now we are trying to access it inside the `btn_printReversed_Click` event handler. Therefore, it is inaccessible. We could resolve this by defining `firstName` in a higher scope: **Class level**. We will define it outside the click event handler methods, as seen in Figure 8. Now, the `firstName` variable has the class-level scope and it should be accessible from any methods defined within the same class where the variable is defined.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
    //Creating the variable with class level scope
    string firstName;

    private void btn1_printFullName_Click(object sender, EventArgs e)
    {
        firstName = txt_firstName.Text; //Store the user's first name

        //Define the variable and store the user's last name
        string lastName = txt_lastName.Text;

        string fullName = firstName + " " + lastName; //Produce the fullName and store it.
        txt_fullName.Text = fullName; //Print the fullname inside the textbox
    }

    private void btn_printReversed_Click(object sender, EventArgs e)
    {
        firstName = txt_firstName.Text;
    }
}
```

Figure 8. Changing the scope of the `firstName` variable.

Please note that, in Figure 8, all occurrences of the `firstName` variable are highlighted with grey background. This is because the cursor is positioned inside one of the `firstName` keywords and Visual Studio highlights all occurrences to help us easily identify where this variable has been used in our code.

We will also define `lastName` at the scope of the class scope and finish the code for `btn_printReversed_Click` event handler. You can see the complete code in Figure 9. Please note that several variables of the same type can be defined in a single line as illustrated in Figure 9.

Such variables defined at class level scope are known as **class fields**. Fields determine the characteristics of a class that should be accessible by the methods of the same class. For example, a *Person* class would have *Name*, *Address*, *Photo*, and *Birthday* fields, and the methods inside the *Person* class may need these fields for different reasons (e.g., sending a personalized birthday message).

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    //Creating the variable with class level scope
    string firstName, lastName;

    private void btn1_printFullName_Click(object sender, EventArgs e)
    {
        firstName = txt_firstName.Text; //Store the user's first name
        lastName = txt_lastName.Text; //Store the user's first name

        txt_fullName.Text = firstName + " " + lastName; //Print the fullname
    }

    private void btn_printReversed_Click(object sender, EventArgs e)
    {
        firstName = txt_firstName.Text; //Store the user's first name
        lastName = txt_lastName.Text; //Store the user's first name

        txt_fullName.Text = lastName + ", " + firstName; //Print the fullname
    }
}
```

Two variables (or fields) are defined in the same line.

Figure 9. Complete code with class level variables defined.

Similarly, we have a `Form1` class, and inside this class we defined two fields, `firstName` and `lastName`. These fields were utilized by different methods (i.e., click event handlers of `btn_printFullName` and `btn_printReversed`) defined inside the `Form1` class.

3) Working with int, double, and decimal variable types

`string` variables can hold only text values. Although they can store numbers, they are stored as text, thus not allowing mathematical computations. To store numeric data types and use them in computations, we can take advantage of 3 primary variable types in C#: `int`, `double`, and `decimal`.

A variable of `int` type can hold a numeric value (positive or negative) such as 5, -67, 0, etc. Numbers with a fractional part (e.g., 13.5) cannot be stored using `int` type variables. The value range that an `int` variable can hold is between -2,147,483,648 and 2,147,483,647. A variable of `int` type occupies 32 bits of memory.

Both `double` and `decimal` variables hold numbers with a fractional part. The main difference between these two type is that while `double` type allows for 15 digits of precision in the fractional part, `decimal` type allows for 28 digits of precision. `decimal` variables are mostly preferred in financial applications that require very precise calculations. As `decimal` is mostly used for storing monetary values, the numeric value should have `m` or `M` appended at the end (e.g., `18.6m`). In other cases, `double` data type is preferred to store fractional numbers as `double` variables occupy less space in the memory (64 bits vs 128 bits).

We will create a new project (named `Week 4_2`) to practice creating and using numeric variables. In this project, change the name and text of the form as suggested in Figure 10.

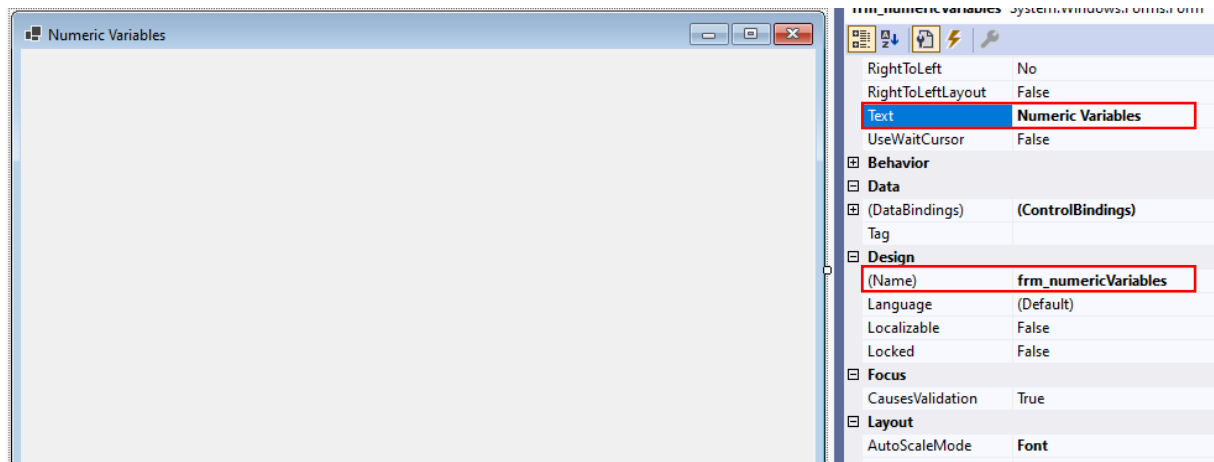


Figure 10. Configuring the form in a new project called `Week 4_2`.

Please double click on the form, to create a form load handler. Inside this handler, we will define three `int` variables to store several data about an educational institution, as shown in Figure 11.

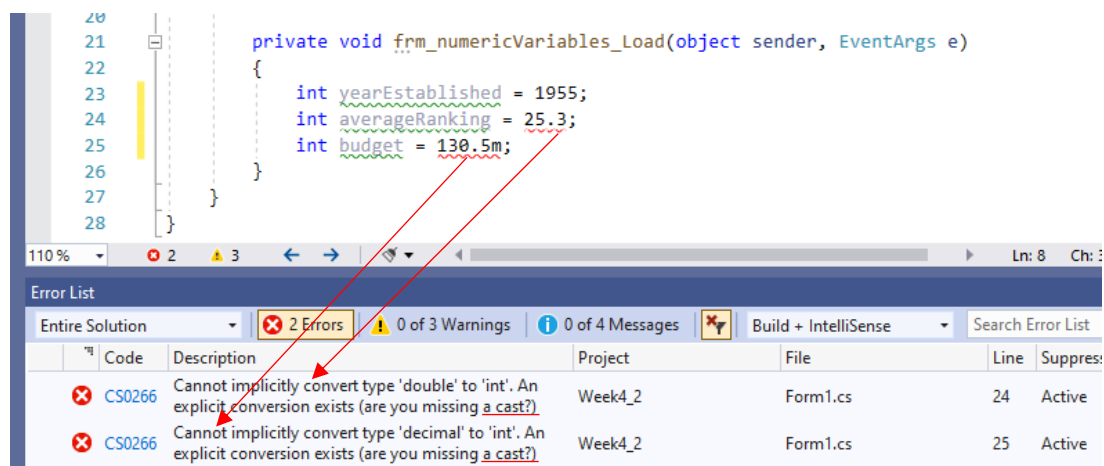


Figure 11. Creating `int` type variables.

Assigning `1955` to the `yearEstablished` variable is successful since `1955` is a number without any fractions. However, assigning the `25.3` (`double` data type) and `130.5m` (`decimal` data type) values were not allowed. The associated errors were listed in the Error List window as indicated in Figure 11. Neither `double` nor `decimal` data types can be assigned to an `int` variable, since this would require the removal

of the fractional part, thus causing information loss. For this reason, the compiler does not allow for this operation and throws (almost the same) errors.

If you read the error messages carefully, you may notice that they suggest a way to resolve the errors: **explicit casting**. Explicit casting is used to communicate to the compiler that we are intentional in assigning `double` nor `decimal` values to an `int` variable and we are aware of possible data loss resulting from this conversion.

For explicit casting, you need to simply add the data type within the parenthesis just before the value that you want to convert, such as `(int) 13.3` or `(decimal) 14`. We will convert the `double` and `decimal` values in our code to `int` using explicit casting as seen in Figure 12.

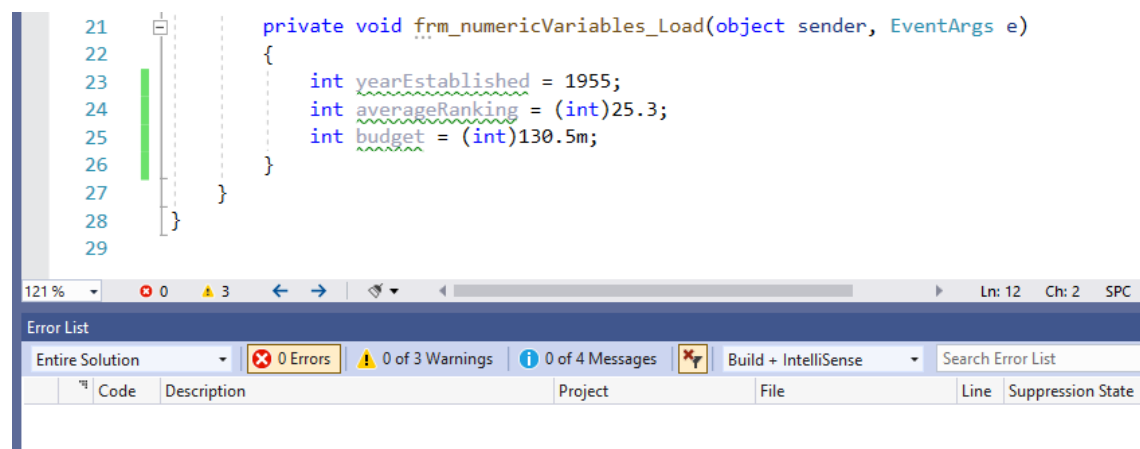


Figure 12. Explicit casting to `int` type.

The errors have disappeared after casting the variables to `int`. However, as mentioned shortly before, this will cause some data loss. To check the values of `averageRanking` and `budget` variables in runtime, we will use **Break Points**. You can add a break point to any line in your code. In runtime, Visual Studio pauses at Break Points and switches to *Break Mode* to allow you to monitor and check the values assigned to variables, which you can use to identify **logical** errors if any.

We will add a break point to the line 26 (before exiting from the form load handler). To do that please click on the grey on the left corresponding to the line 26 as indicated in Figure 13.

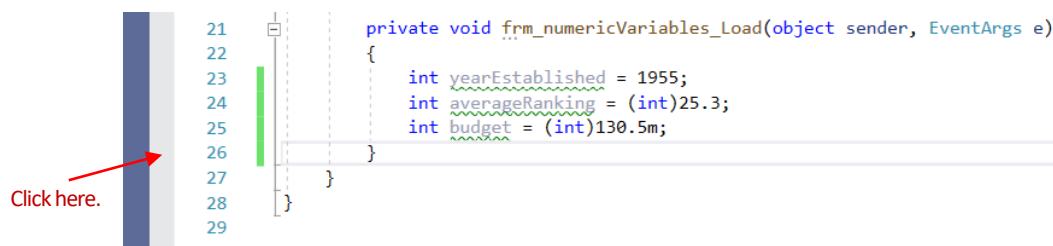


Figure 13. Adding a break point.

After a single click, the break point should be added. A red circle will indicate the break point as shown in Figure 14.

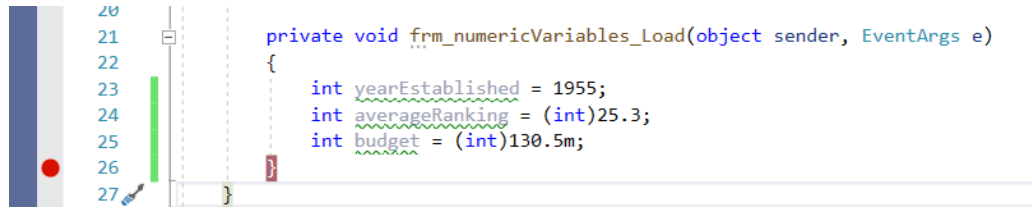


Figure 14. Break point added.

Now, please run the application with debugging (press F5). Break points are activated only in debugging mode. The application should pause at the break point, as seen in Figure 15.

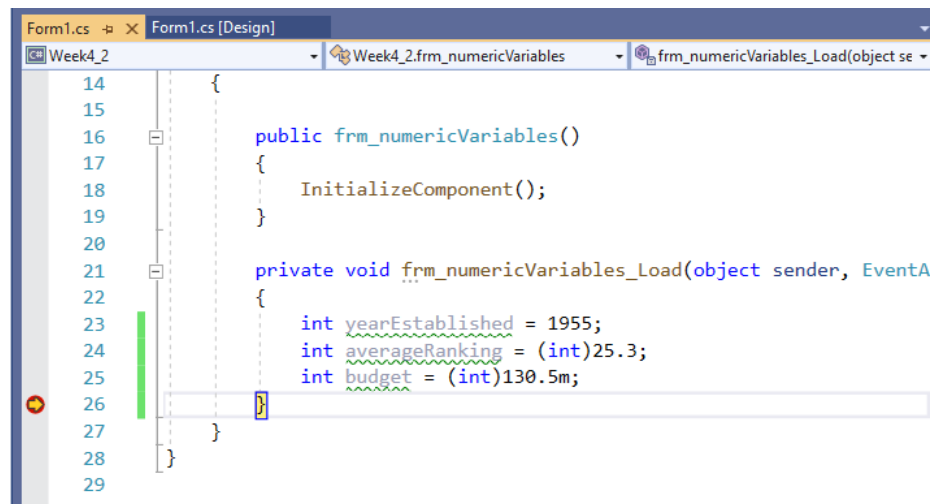


Figure 15. Application pauses at break point.

If you mouse over any of the variable, its current value should be displayed in a little popup window as shown in Figure 16.

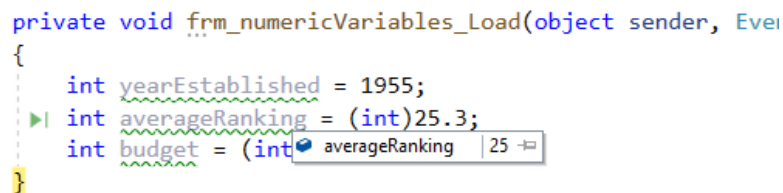


Figure 16. Popup window showing the value of averageRanking.

Alternatively, you can do a write click on each of the variables and then choose the **Add Watch** option from the menu.

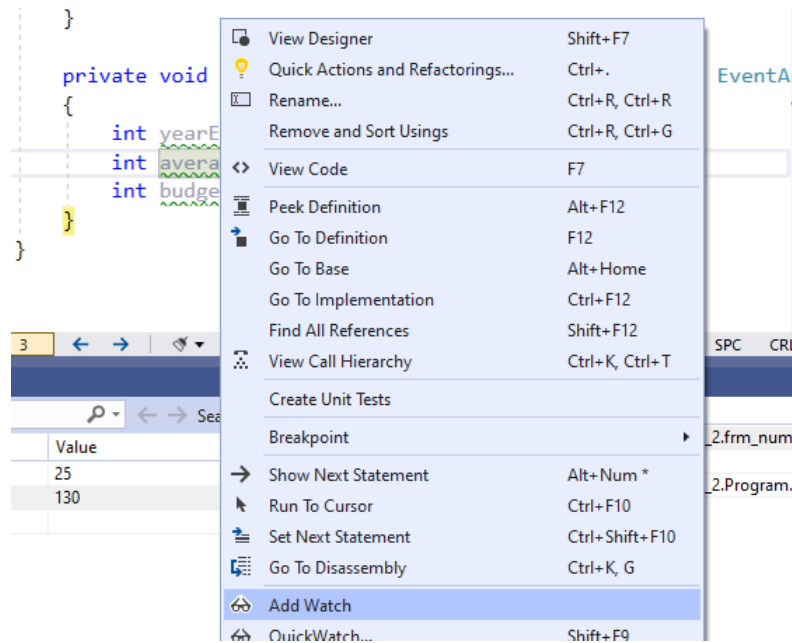


Figure 17. Add watch option.

Once both variables are added to watch, you can see their values at break point using the **Watch** window as shown in Figure 18 below.

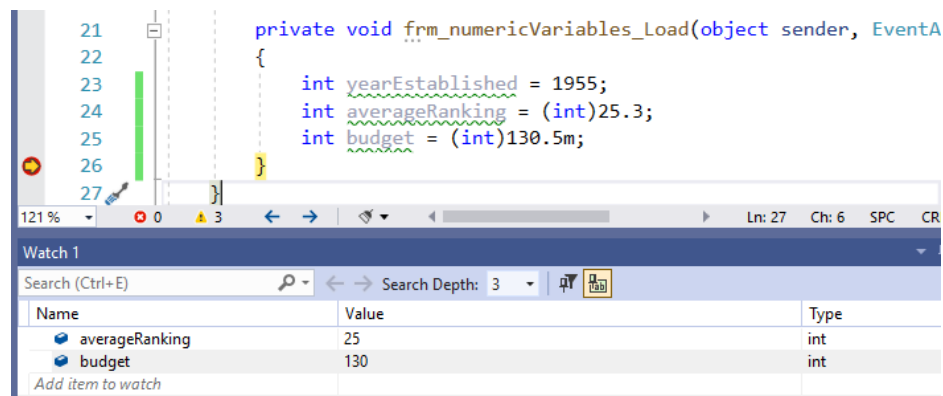


Figure 18. Watch list window.

As you see in Figure 18, `averageRanking` has the value of 25 and `budget` has the value of 130. That is, the fractional parts were removed from 25.3 and 130.5m and the new values were stored as `int`. We will continue with more examples. Figure 19 and Figure 20 show example codes in which this time the same variables were defined as `double` and `decimal` data types.

```
private void frm_numericVariables_Load(object sender, EventArgs e)
{
    double yearEstablished = 1955;
    double averageRanking = 25.3;
    double budget = 130.5m;
}
```

Figure 19. Creating `double` variables.

```
private void frm_numericVariables_Load(object sender, EventArgs e)
{
    decimal yearEstablished = 1955;
    decimal averageRanking = 25.3;
    decimal budget = 130.5m;
}
```

Figure 20. Creating `decimal` variables.

As you may notice, assigning a `decimal` value (*130.5m*) to a `double` variable (*budget*) is not allowed. Similarly, assigning a `double` value (*25.3*) to a `decimal` variable (*averageRanking*) causes an error. This is because of possible data loss (each data type has a different precision). On the other hand, assigning an `int` value (*1995*) does not cause any error since it does not entail any data loss.

I leave it to you as an exercise to cast these values and check their values after casting using break points.

4) Converting between numeric and string data types

Often we need to make conversion between numbers and strings. For example, if you collect a numeric data (e.g., *15*) from users through the textbox control, this data is automatically stored as `string` ("*15*") inside the **Text** property of the textbox. To be able to use this value in mathematical operations, you would need to convert it to the desired numeric data type (`int`, `double` or `decimal`).

Explicit casting is **NOT** allowed for conversions between numeric and string data types. Instead, we need to use **Parse** methods of the numeric data types:

- `int.Parse()` converts a `string` to an `int` → `int.Parse("10")`
- `double.Parse()` converts a `string` to a `double` → `double.Parse("21.5")`
- `decimal.Parse()` converts a `string` to a `decimal` → `decimal.Parse("236.23")`

For the `Parse` methods you need to provide a `string` value inside the parentheses, known as **argument**, that you want to convert. The `Parse` method returns the converted value.

We will explore the use of `Parse` method and some other related concepts in a new application. The goal of this application is to calculate a student's engagement level in a course. This calculation will be made using a formula that consists of several engagement indicators. These indicators are:

- Missed classes (*mc*) : Number of weeks that student missed the lectures.
- Percentage of readings made (*rm*) : Percentage of reading materials that were completed.
- Average quiz attempts (*qa*) : Average number of attempts in all quizzes.
- Average quiz scores (*qs*) : Average score across all quizzes.
- Discussion posts (*dp*) : Total number of discussion posts made throughout the course.

The formula for calculating the engagement level is:
$$\frac{rm \times 60 + qa \times 10 + qs \times 15 + dp \times 15}{mc \times 100}$$

The graphical user interface for the application is provided in Figure 21. Please create a similar design and use the suggested control names in the figure.

Figure 21 shows a Windows application window titled "Numeric Variables". Inside the window, the title "Student Engagement Calculator" is centered. Below the title, there are five input fields, each preceded by a label: "Missed classes :", "Average quiz attempts :", "Average quiz score :", "Discussion posts :", and "% of readings made :". To the right of each input field, a red arrow points to a text label: "txt_missedClasses", "txt_avgQuizAttempt", "txt_avgQuizScore", "txt_discussionPosts", and "txt_readingsMade" respectively. Below these input fields, there is a "Calculate" button. A red arrow points from the button to the label "btn_calculate" below it.

Figure 21. The interface for the Student Engagement Calculator application.

Clicking `btn_calculate` should apply the engagement indicators (provided by the user) in the formula for calculating the student engagement level. Therefore, most of the code will go inside the click event handler of `btn_calculate`.

Please double click on `btn_calculate` to automatically create its click event handler. We will first read the indicator values from the textboxes and assign them to variables with proper data type. Figure 22 shows the example code where each value is converted to a proper data type using the `Parse` method and stored in a variable of that type. Indicators that might have fractional part are converted to `double` type and assigned to a `double` variable. The others are converted to `int` type and stored in an `int` variable.

```
private void btn_calculate_Click(object sender, EventArgs e)
{
    int mc = int.Parse(txt_missedClasses.Text);
    double rm = double.Parse(txt_readingsMade.Text);
    double qa = double.Parse(txt_avgQuizAttempts.Text);
    double qs = double.Parse(txt_avgQuizScore.Text);
    int dp = int.Parse(txt_discussionPosts.Text);
}
```

Figure 22. Reading engagement indicators as provided by the user.

After getting these values, now we are ready to put them in the formula and calculate the final output. As you may remember the formula has **Numerator** and **Denominator** parts. We will compute these values separately as shown in Figure 23. In the computation, we will use the *addition operator* `+` to add values, and *multiplication operator* `*` to multiply values.

```
private void btn_calculate_Click(object sender, EventArgs e)
{
    int mc = int.Parse(txt_missedClasses.Text);
    double rm = double.Parse(txt_readingsMade.Text);
    double qa = double.Parse(txt_avgQuizAttempts.Text);
    double qs = double.Parse(txt_avgQuizScore.Text);
    int dp = int.Parse(txt_discussionPosts.Text);

    double numerator = rm * 60 + qa * 10 + qs * 15 + (double)dp * 15;
    double denominator = mc * 100;
}
```

Figure 23. Computing the numerator and denominator values.

Next, we will use the *division operator* / to divide the numerator by denominator, assign the resulting value to a `double` variable called `engagementLevel`. We will use `MessageBox.Show()` method to display the final outcome. This method prints the provided string in a popup window. Before printing `engagementLevel` we have to convert it to string using the `ToString()` method. The details of the code are shown in Figure 24.

```
double numerator = rm * 60 + qa * 10 + qs * 15 + (double)dp * 15;
double denominator = mc * 100;

double engagementLevel = numerator / denominator;

MessageBox.Show(engagementLevel.ToString());
}
```

Figure 24. Computing the engagement level and printing it.

Now you can test your application by pressing Ctrl+F5. Figure 25 shows an example screen from the running application.

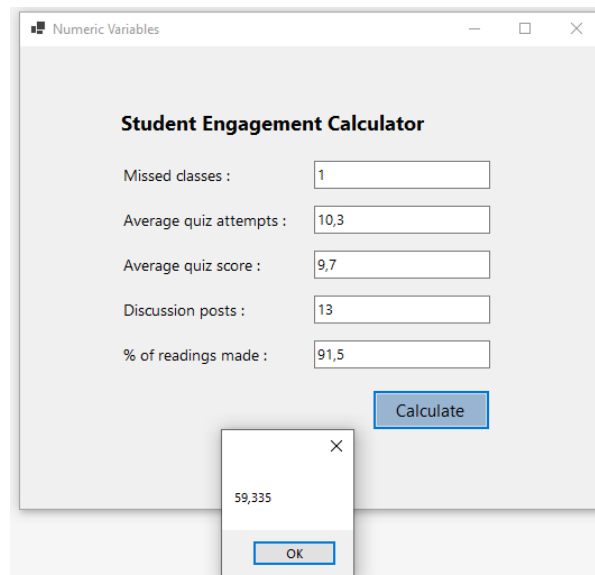


Figure 25. Testing the application.

5) Handling exceptions

Often our code results in unexpected errors in runtime while the users use the application. These errors are called exceptions. If the program throws an exception, the application terminates, and a technical detailed report about the error is provided to help us (the programmers) resolve it. These error reports have no use for the end-users, and they are targeted at the programmers. Therefore, it is better that we show a custom and less scary error message to the users and let them continue using the application while keeping the technical details to ourselves as the programmers.

To be able to do so, C# provides an exception handling mechanism using `try-catch` statement. This statement is composed of two parts: `try` block, where any statement may cause an error, and `catch` block, where the potential error occurred in the `try` block is handled.

A rather simple `try-catch` statement is shown below. First, the statements inside the `try` block will be executed in a linear order. If an exception is thrown in any line, the program will jump to the `catch` block and execute the code in there. In the following code, an exception will be thrown at line 4 since the string "15abc" cannot be converted to an `int`. Then, the program will jump inside the `catch` block and will print an error message. If no exceptions are thrown inside the `try` block, then the `catch` block is skipped.

```
1      try
2      {
3          int[] number1 = 20;
4          int[] number2 = (int) "15abc";
5          int[] number3 = (int) "15";
6      }
7      catch
8      {
9          MessageBox.Show("Input is invalid");
10     }
```

When an exception is thrown, an `Exception` object is created in the memory. Detailed information about the error is automatically stored in an `Exception` object. This object can be accessed in the `catch` statement by passing an `Exception` type object. We updated the previous code by adding a new parameter inside the `catch` statement, as you can see below. The `Message` property of the `Exception` object contains a short description of the error. The following code prints this property to provide a bit more specific information about the error.

```
1      try
2      {
3          int[] number1 = 20;
4          int[] number2 = (int) "15abc";
5          int[] number3 = (int) "15";
6      }
7      catch(Exception ex)
8      {
9          MessageBox.Show(ex.Message);
10     }
```

Now, we will improve our application by adding try-catch statements to catch any exceptions regarding invalid user inputs. Our application asks for only numeric inputs; however, sometimes users may enter an invalid number, which will halt the program. Instead, we should display a custom message to warn the users and allow them re-entering the value without quitting from the program.

First, we will create a try-catch statement inside the click event handler and place all existing code inside the try block as shown in Figure 26.

```
try
{
    int mc = int.Parse(txt_missedClasses.Text);
    double rm = double.Parse(txt_readingsMade.Text);
    double qa = double.Parse(txt_avgQuizAttempts.Text);
    double qs = double.Parse(txt_avgQuizScore.Text);
    int dp = int.Parse(txt_discussionPosts.Text);

    double numerator = rm * 60 + qa * 10 + qs * 15 + (double)dp * 15;
    double denominator = mc * 100;

    double engagementLevel = numerator / denominator;

    MessageBox.Show(engagementLevel.ToString());
}
catch (Exception ex)
{
}
```

Figure 26. Adding the try-catch statements.

Now, we will write the necessary code for the catch block to display a custom error message. To do so, we will use a string builder. The first part of the message will say something rather generic. Then, we will append the more detailed error message at the end. The complete code for the catch block is given in Figure 27.

```
catch (Exception ex)
{
    StringBuilder sb_error = new StringBuilder();

    sb_error.AppendLine("Please enter a valid numeric value in all fields.");
    sb_error.AppendLine();
    sb_error.AppendLine("More info:");
    sb_error.AppendLine(ex.Message);

    MessageBox.Show(sb_error.ToString());
}
```

Figure 27. Providing a custom error message inside the catch block.

Now, you can run your application. Test it with an invalid entry. You should get the following error message as shown in Figure 28.

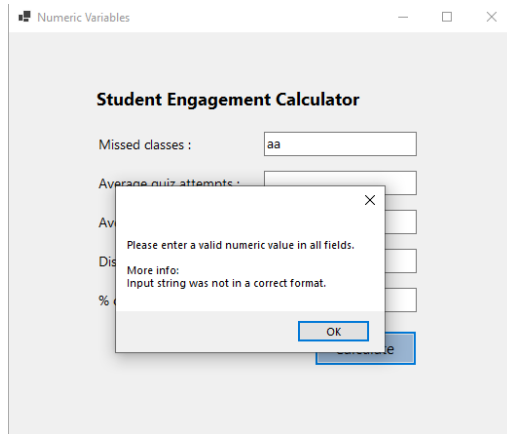


Figure 28. Custom error message is displayed.