

Module 5 – Decision structures.

Sometimes we have to *decide* to execute or skip a specific piece of code under certain conditions depending on what we are intending to achieve with our application. For example, in an online course, students might be allowed to access a resource only if they scored above the threshold in a prerequisite quiz. In this case, we may want to control if the quiz score is above the threshold (that is, *checking if the condition meets or not*) and then *decide* the next action: execute the code that allows the students access the target resource or execute the code that shows an error message and redirect the student to the quiz page.

1) If statement

Such condition-based control structures are called decision structures. In C#, like in many other programming languages, `if` statement is used to create decision structures. The syntax of `if` statements follow this structure:

```
if (condition statement)
{
    code...
}
```

After the `if` keyword, inside the parenthesis goes the condition statement. Condition statement must be a *Boolean* expression which returns either `true` or `false`. If the condition check results in a `true` value, then the code inside the braces `{ }` is executed, if not, it is skipped. In other words, the code inside the braces is *conditionally* executed.

We use **relational operators** to create the Boolean expressions in `if` conditions. These operators help us identify if a certain type of relationship exists between two values. These operators are provided in Table 1 below. Examples for each operator is also provided in the table.

Table 1. Relational operators

| Operator | Meaning | Example #1 | Example #2 |
|--------------------|--------------------------|---|--|
| <code>==</code> | Equal to | <code>10 == 12</code> returns <code>false</code> . | <code>12 == 12</code> returns <code>true</code> . |
| <code>!=</code> | Not equal to | <code>10 != 12</code> returns <code>true</code> . | <code>12 != 12</code> returns <code>false</code> . |
| <code>></code> | Greater than | <code>10 > 12</code> returns <code>false</code> . | <code>12 > 12</code> returns <code>false</code> . |
| <code><</code> | Less than | <code>10 < 12</code> returns <code>true</code> . | <code>12 < 12</code> returns <code>false</code> . |
| <code>>=</code> | Greater than or equal to | <code>10 >= 12</code> returns <code>false</code> . | <code>12 >= 12</code> returns <code>true</code> . |
| <code><=</code> | Less than or equal to | <code>10 <= 12</code> returns <code>true</code> . | <code>12 <= 12</code> returns <code>true</code> . |

The `==` operator (please do not get confused with the *assignment operator* `=`) checks if the value on its left is **equal** to the value on its right. If the values are the same then the expression is evaluated as `true`, otherwise `false`. The `!=` operator is the opposite of `==`. It returns `true` if the values are NOT the same. Let's assume that we have an `int` variable called `grade`, which has the value of 85. The expression `grade == 85` would be `true`, `grade != 85` would be `false`.

The `>` operator determines if the value on its left is greater than the value on its right. Similarly, the `<` operator checks if the value on the left is greater than the value on the right. For example, `grade > 85` would be evaluated as false, `grade < 90` would be evaluated as true. We can use `>=` and `<=` operators to check for two relationships at the same time: the value on the left is greater/less than **or** equal to the value on the right. For instance, `grade >= 85` would be evaluated as true since the value of `grade`, 85, is equal to the value compared, 85 (although it is not greater).

Now we will practice what we have learned by creating a new application. Please create a new project with a name `Week5_1` or any other name you prefer. This application will allow for setting a threshold as a success or failure criteria in a course. If the entered student grade is higher than or equal to the threshold, we will print the success message otherwise, we will print the failure message.

We will use the **GroupBox** control to visually group the associated controls inside a box as shown in Figure 1. GroupBox control is included in the *Containers* section of Toolbox. Any control you drop over a GroupBox control is contained inside the GroupBox control. When you move GroupBox, all the contained controls also move together. GroupBox also provides a nice thin border and a title that makes it easier for users to visually understand your interface. As shown in Figure 1, please add a GroupBox control, and inside it, add a label, a textbox and a button.

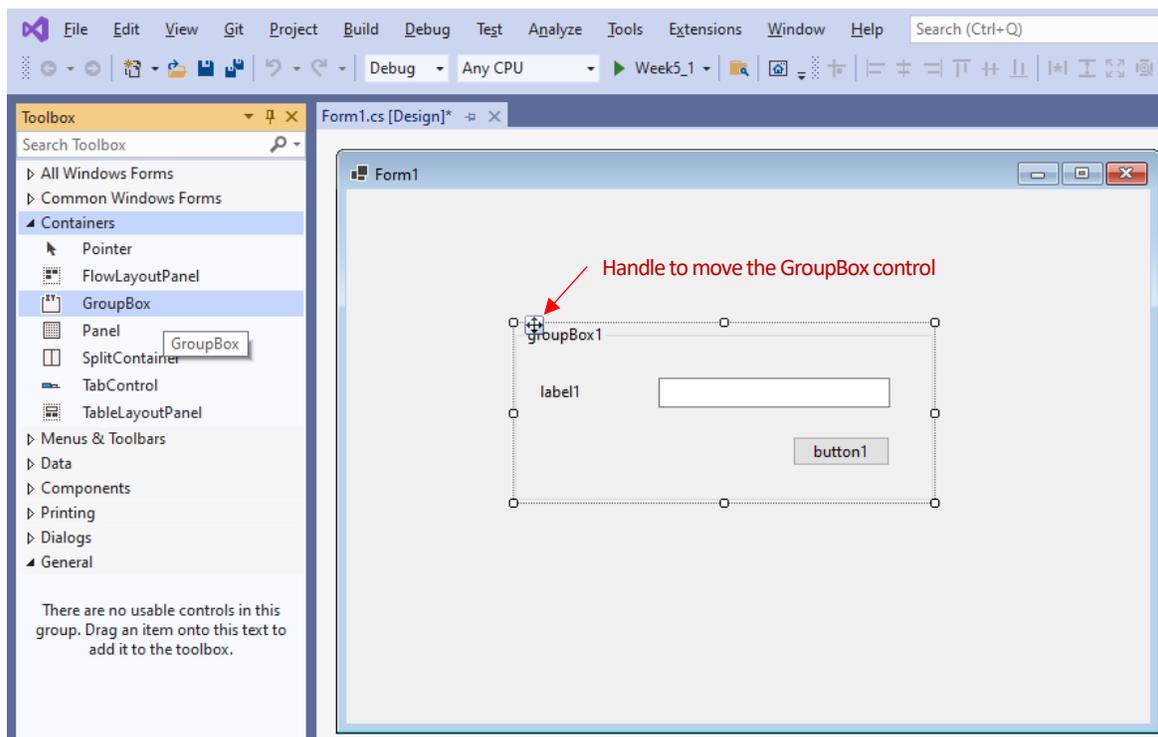


Figure 1. Adding the controls for the threshold part.

After adding the controls, please update their Text/Name properties as suggested in Figure 2.

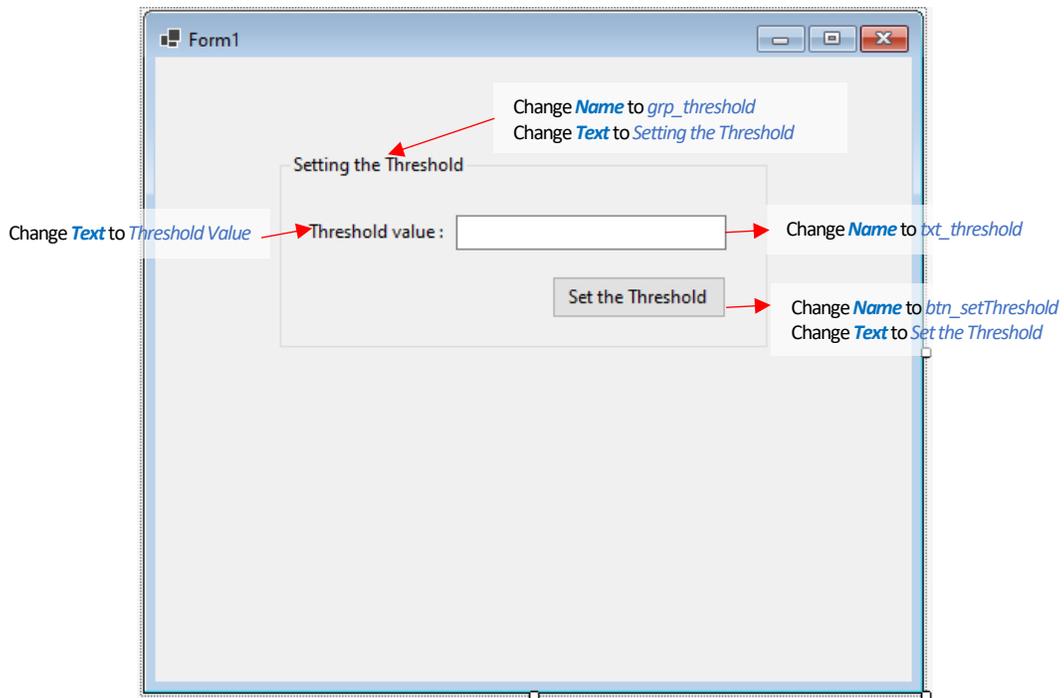


Figure 2. Updating the controls for the threshold part.

We need another GroupBox control to design the section where the user will enter the student grade and check if the grade is higher/lower than the threshold. We could create a new one following the same steps as we already did for the Threshold GroupBox. Instead, to save time, we will duplicate the existing GroupBox control. To do that, please select the GroupBox control, and in your keyboard press Ctrl+C and then Ctrl+V. Move the duplicated control to the down to obtain the following interface shown in Figure 3.

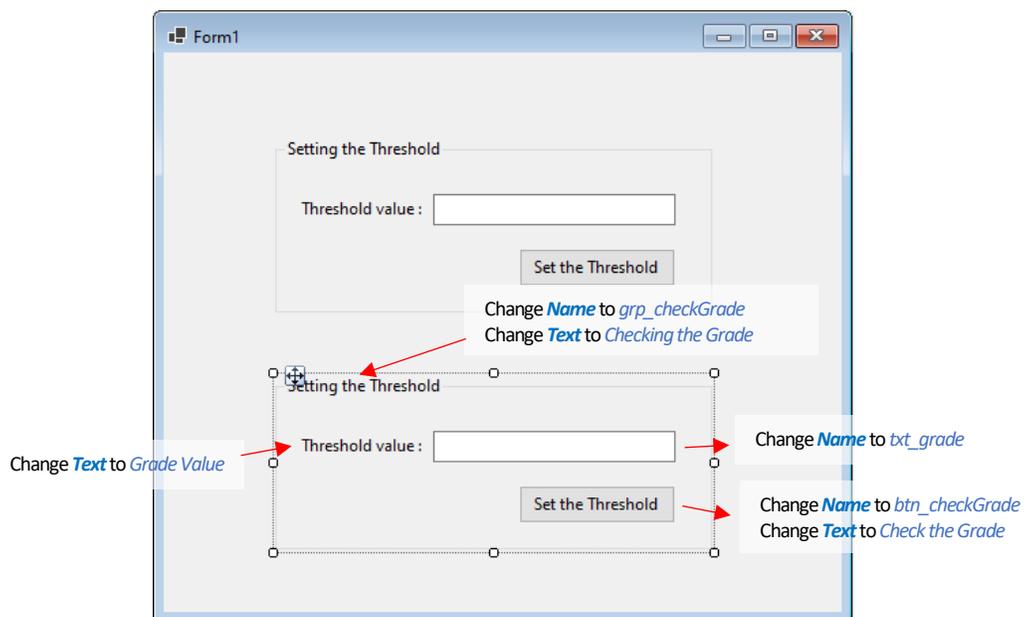


Figure 3. Duplicating the threshold GroupBox control.

Now, all we need to do is to update the duplicated controls. Please update the Name/Text values of the controls as suggested in Figure 3. The final interface should look like the form in Figure 4.

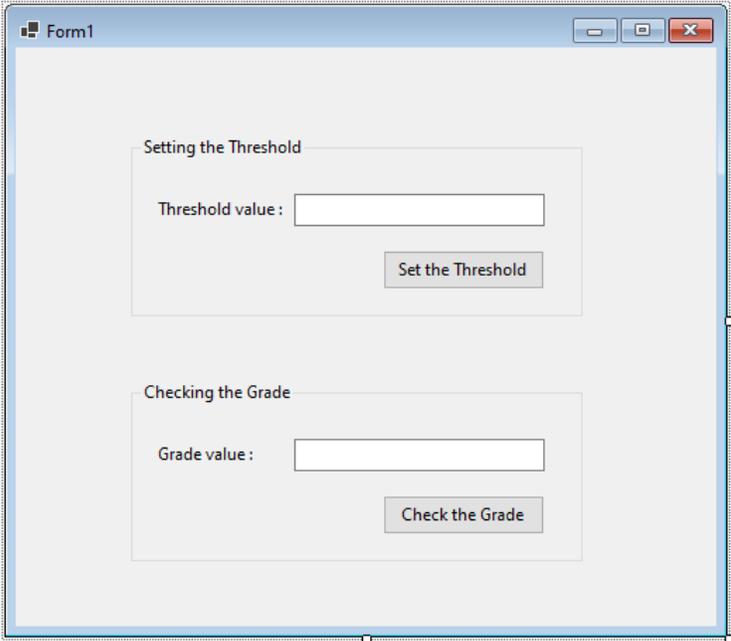


Figure 4. The finalized design of the form.

Now, we are ready to start coding. We will write the code for the click event handler of *the Check the Grade* button (`btn_checkGrade`). Please double click on this button to automatically create its click event handler. We need to know the values typed into the threshold textbox (`txt_threshold`) and the grade textbox (`txt_grade`). To use these values, we better define two `int` variables to store these values.

As you may see in Figure 5, we declared two variables of `int` type named `threshold` and `grade`. These variables are underlined with green line since they have not been used yet. This is a very useful visual clue to identify unused or unnecessary variables in our code.

```
namespace Week5_1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void btn_checkGrade_Click(object sender, EventArgs e)
        {
            int threshold, grade;
        }
    }
}
```

Figure 5. Defining the int variables.

You may remember that the text typed into a textbox are stored in its **Text** property. We will access the values entered into `txt_threshold` and `txt_grade` through the **Text** property and assign these values to the `threshold` and `grade` variables respectively, as shown in Figure 6.

```
private void btn_checkGrade_Click(object sender, EventArgs e)
{
    int threshold, grade;
    threshold = txt_threshold.Text;
    grade = txt_grade.Text;
}
```

Figure 6. Reading the text values into the int variables.

However, assigning the values to the variables causes errors as indicated by the red underlines in Figure 6. The reason for error is that we are trying to assign a **string** value to a variable of **int** type, which are incompatible. We need to convert the `Text` values to **int** data type before assigning them to the variables. We will use the **Parse** method of the **int** type as we learned in the previous chapter. Please see Figure 7 for the sample code.

```
private void btn_checkGrade_Click(object sender, EventArgs e)
{
    int threshold, grade;
    threshold = int.Parse(txt_threshold.Text);
    grade = int.Parse(txt_grade.Text);
}
```

Figure 7. Converting the text values into int data type and assigning them to variables.

Now that we have the `threshold` and `grade` values as **int**, we can move to the final part where we need to identify if the `grade` is greater than the `threshold` and then decide to print the “Success” message (or not). As you may expect, we will write an **if** statement as seen in Figure 8.

```
private void btn_checkGrade_Click(object sender, EventArgs e)
{
    int threshold, grade;
    threshold = int.Parse(txt_threshold.Text);
    grade = int.Parse(txt_grade.Text);

    if(grade >= threshold)
    {
        MessageBox.Show("Success!");
    }
}
```

Figure 8. Writing the if statement to compare the grade and threshold values.

The code inside the braces `{ }` of the **if** statement prints the “Success” message in a popup window. This line of code will be executed only if the `grade` is equal to or greater than the `threshold`.

We will test our code with debugging mode enabled. Please add a breakpoint at the line 28 and press F5 to run the application with debugging enabled. See Figure 9 for the details.

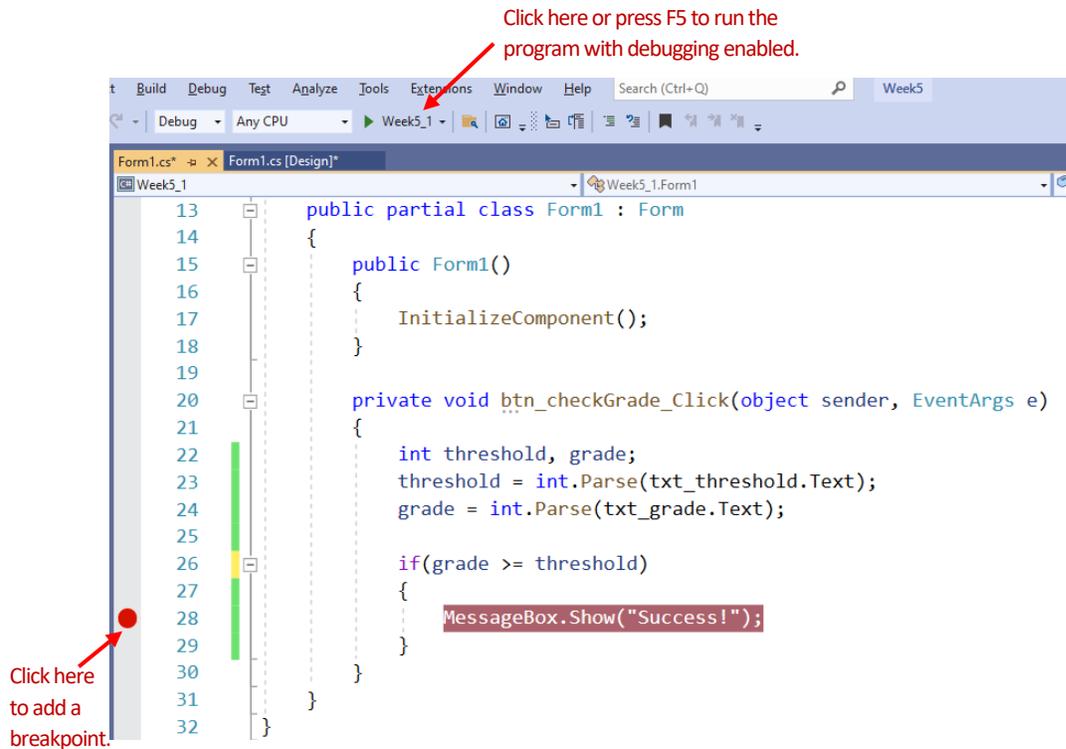


Figure 9. Adding a breakpoint to pause inside the if statement.

In the running application, please enter 60 for the threshold field and 65 for the grade field and then press the “**Check the Grade**” button, as shown in Figure 10.

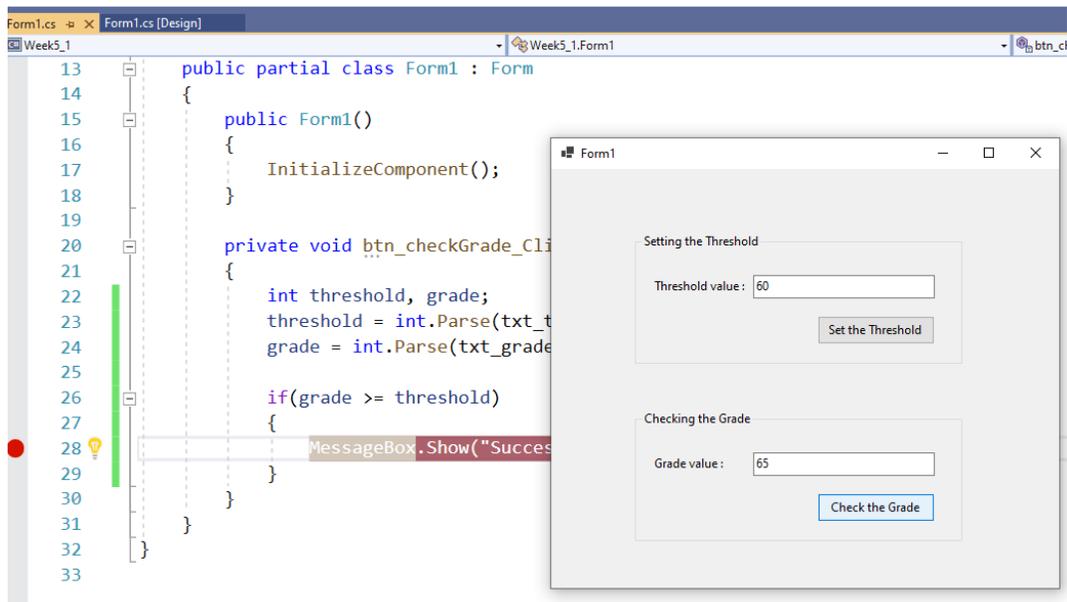


Figure 10. Adding a breakpoint to pause inside the if statement.

The program should pause at the breakpoint as seen in Figure 11, which means that the Boolean expression (`grade >= threshold`) inside the `if` condition returned `true` since the grade value provided was greater than the threshold value.

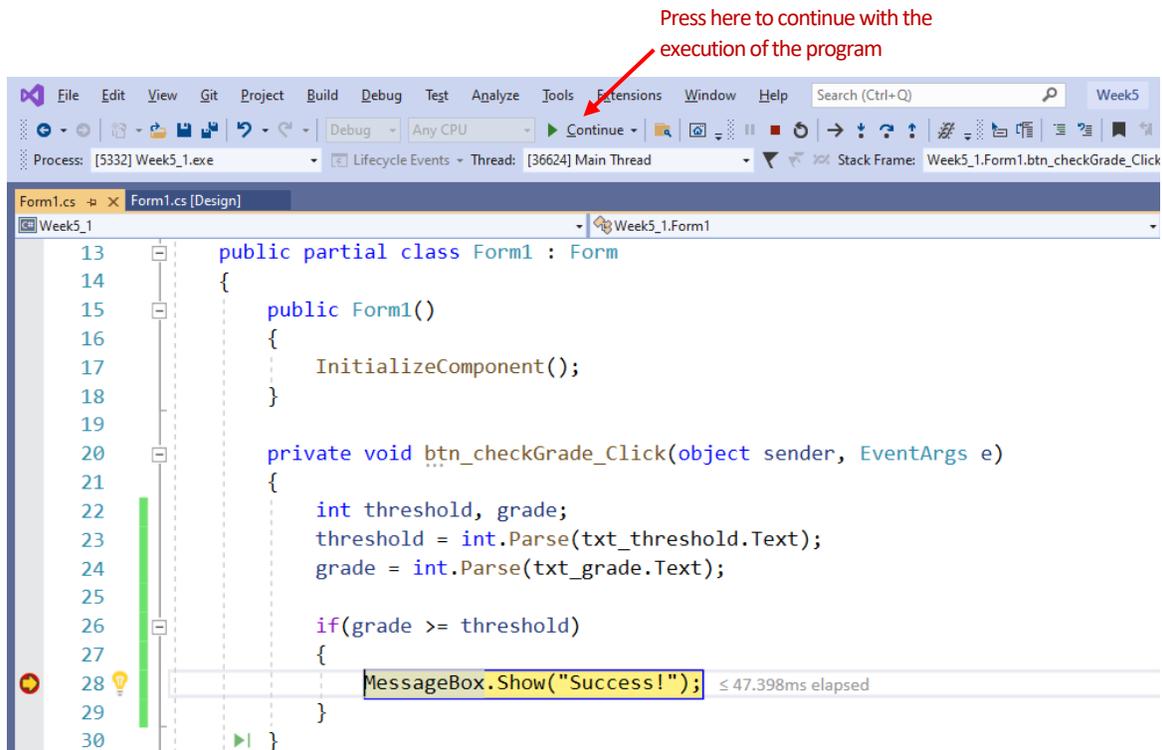


Figure 11. Program paused at the breakpoint.

Please click on **Continue** (as shown in Figure 11) to continue with the execution of the program. The form should be displayed again, and the popup window with the Success text should appear as in Figure 12.

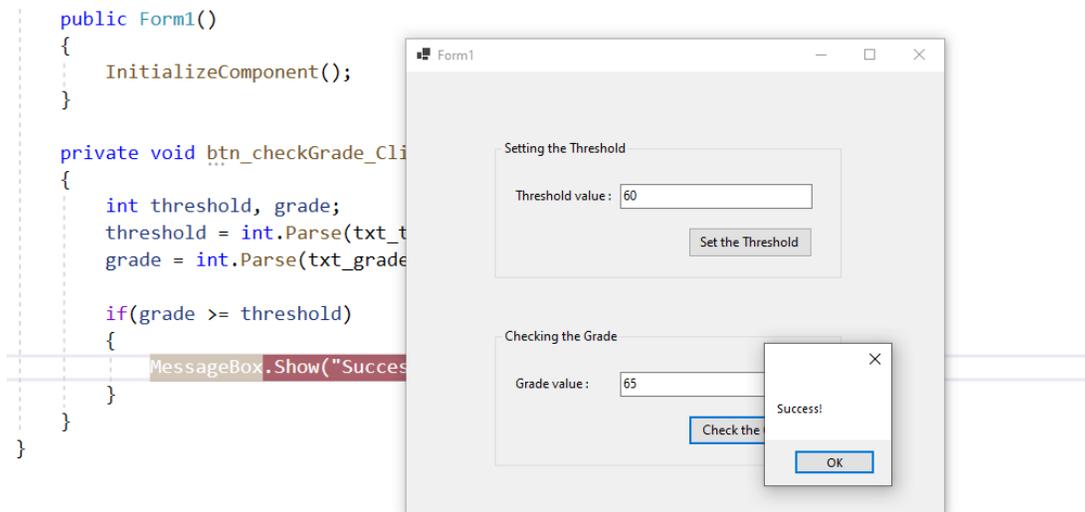
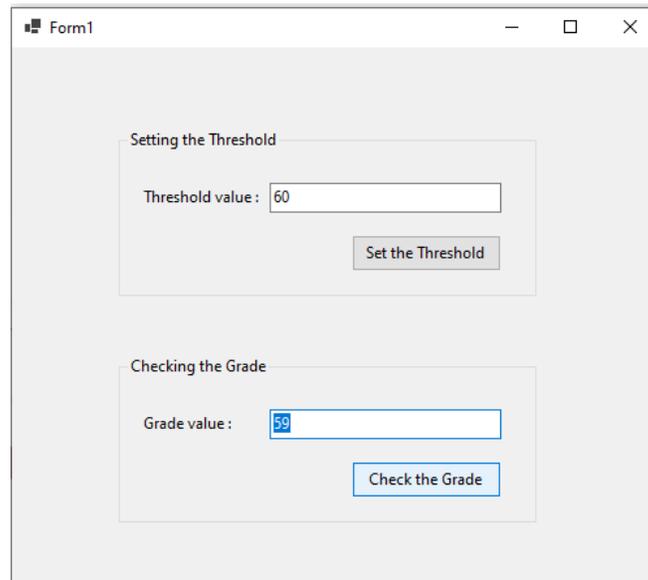


Figure 12. The Success message is displayed.

We will try another grade value that is smaller than the threshold. Please enter 59 in the grade field, and then press the Check the Grade button (see Figure 13).



The screenshot shows a window titled "Form1" with two distinct sections. The top section, titled "Setting the Threshold", contains a text box labeled "Threshold value:" with the number "60" entered, and a button labeled "Set the Threshold". The bottom section, titled "Checking the Grade", contains a text box labeled "Grade value:" with the number "59" entered, and a button labeled "Check the Grade".

Figure 13. Trying a smaller grade value.

Nothing has happened! Why do you think clicking the button did not have any effect?

2) If-else statements

Decision structures are often more complex, and they involve multiple alternative paths, which implies checking for multiple conditions. In the example above, we perform some operations only if the grade is *greater* than the threshold. How about showing a different message if the grade is lower? In such cases where we need to check two conditions, we can use *if-else* statement. The basic format of *if-else* statement has the following structure:

```
if (Boolean expression)
{
    code... ← Executed if Boolean expression is TRUE.
}
else
{
    code... ← Executed if Boolean expression is FALSE.
}
```

The first part is the typical *if* statement that we have learned already. Afterwards comes the *else* statement, where we include the code to be executed when the Boolean expression in the *if* statement is *FALSE*. That is, if the Boolean expression is *true*, then the code included in the *else* block is neglected. If the Boolean expression is *false*, the code included in the *if* block is skipped, and all code inside the *else* block is executed.

We will use the `if-else` statement in our application to print a “Failure” message when the grade is NOT greater than the threshold. We will update our code as shown in Figure 14.

```
private void btn_checkGrade_Click(object sender, EventArgs e)
{
    int threshold, grade;
    threshold = int.Parse(txt_threshold.Text);
    grade = int.Parse(txt_grade.Text);

    if (grade >= threshold)
    {
        MessageBox.Show("Success!");
    }
    else
    {
        MessageBox.Show("Failure!");
    }
}
```

Figure 14. Changing to if-else statement.

Please run your application and enter a grade value that is smaller than the threshold. As you may expect, the failure message should be displayed when you click Check the Grade button (see Figure 15).

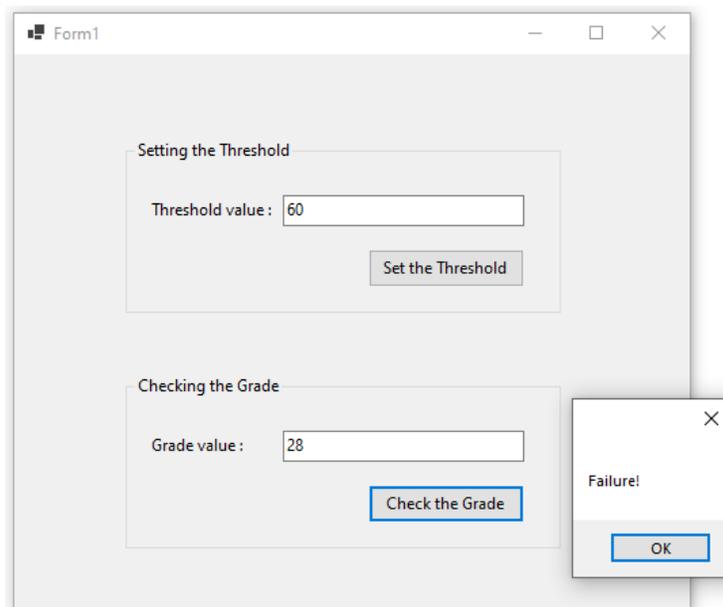


Figure 15. Displaying the failure message.

Please note that we could change the Boolean expression to `grade < threshold` to first check for the condition if the grade is less than the threshold and print the Failure message, then we can include the Success message in the `else` statement. This would change only the structure of `if-else` statement and would not impact the result. Please see Figure 16.

```

private void btn_checkGrade_Click(object sender, EventArgs e)
{
    int threshold, grade;
    threshold = int.Parse(txt_threshold.Text);
    grade = int.Parse(txt_grade.Text);

    if (grade < threshold)
    {
        MessageBox.Show("Failure!");
    }
    else
    {
        MessageBox.Show("Success!");
    }
}

```

Figure 16. Changing the if-else statement.

3) Using TryParse to convert without errors

To check whether a string (such as Text value of a textbox) holds a value that can be converted to a numeric type, we can use the **TryParse** method. This method returns false if the conversion is not possible, without throwing any exceptions. If the conversion is possible, it returns true and holds the converted value in a separate variable. Each of the numeric data types, int, double, and decimal, has TryParse method to convert a string to any of these numeric data types.

TryParse method accepts two mandatory arguments: (1) the string value to be converted, and (2) the name of the variable in which the converted value will be stored. For example, the following code shows how to use the **int.TryParse** method to convert `txt_threshold.Text` into an integer.

```
int.TryParse(txt_threshold.Text, out int threshold)
```

Pay attention to the **out** keyword appearing before the `threshold` variable above. This keyword indicates that `threshold` is an output variable in which a value will be assigned inside the method before the execution of the method terminates. In our case, if the conversion is successful, in the `threshold` variable, the converted value of `txt_threshold.Text` will be stored. If the conversion fails, the `threshold` variable will set to 0.

Since **TryParse** returns either true or false, we can assign it to a Boolean variable named `isSuccessful` as shown below:

```
int isSuccessful = int.TryParse(txt_threshold.Text, out int threshold);
```

`isSuccessful` will be false if `txt_threshold.Text` holds a value that cannot be converted to an integer. `isSuccessful` will be true if the `txt_threshold.Text` value is converted to int type, which then will be stored inside the output variable `threshold`. Then, we can use `isSuccessful` inside and if-else structure to decide whether performing some operations with `threshold` or printing an error message. The code is shown below:

```

if(isSuccessful == true)
{
    //Perform some operations using threshold
}
else
{
    //Print an error message
}

```

Instead of defining an additional Boolean variable, we can directly include the method call inside the if statement, as shown below:

```

if(int.TryParse(txt_threshold.Text, out int threshold))
{
    //Perform some operations using threshold
}
else
{
    //Print an error message
}

```

4) Nested if-else statements

Sometimes, execution of a conditional statement may depend on a higher-level conditional statement. For example, we may want to check if `txt_threshold` holds a valid number, and then if so, we may check if `txt_grade` holds a valid number, and then perform some operations involving both numbers. That is, the second if-else statement **neests** inside the first one, and its execution depends on the first if-else statement. The structure of this nested if-else statements is shown below.

```

if(txt_threshold holds a valid number)
{
    if(txt_grade holds a valid number)
    {
        //Continue with the operations involving both numbers
    }
    else
    {
        //Tell user that txt_grade is invalid
    }
}
else
{
    //Tell user that txt_threshold is invalid
}

```

As covered in the previous subsection, we will use **TryParse** method to check if the textboxes contain a value that can be converted to a number. Below is the updated code with TryParse methods.

```

if(int.TryParse(txt_threshold.Text, out int threshold))
{
    if (int.TryParse(txt_grade.Text, out int grade))
    {
        //BOTH Text values are converted to int successfully,
        //stored inside the threshold and grade output variables
        //The code to perform operations with these values goes here
    }
    else//txt_grade.text is not valid
    {
        MessageBox.Show("Invalid grade value.");
    }
}
else//txt_threshold is not valid
{
    MessageBox.Show("Invalid threshold value.");
}

```

Figure 17. Nested if-else statement to ensure textboxes hold valid numeric values.

As indicated in the Figure 17, we can place our code inside the second `if` block to perform further operations with the `grade` and `threshold` variables. We will compare `grade` and `threshold` to print Failure or Success messages, as shown in Figure 18.

```

if(int.TryParse(txt_threshold.Text, out int threshold))
{
    if (int.TryParse(txt_grade.Text, out int grade))
    {
        //BOTH Text values are converted to int successfully,
        //stored inside the threshold and grade output variables
        //The code to perform operations with these values goes here

        if (grade < threshold)
        {
            MessageBox.Show("Failure");
        }
        else
        {
            MessageBox.Show("Success");
        }
    }
    else...
}
else...

```

Figure 18. Complete code for using nested if-else statements to compare grade and threshold.

Any code that is written in the rest of the document should be placed inside the second if condition and replace the existing if-else statement comparing grade and threshold.

4) else if statements

Often the decision structures involve more than two paths. For example, instead of classifying students' performance as success or fail, we generally assign a letter grade between A and F (depending on the grading schema), which means we need to check for 6 different conditions. If-else statements can allow for more than 2 conditions by adding `else if` statements. `else if` statements should go

between the `if` and the last `else` statements. There can be as many `else if` statements as we need. The format of `else if` statements is shown below:

```
if (first Boolean expression)
{
    code...
}
else if (second Boolean expression)
{
    code...
}
...
else if (nth Boolean expression)
{
    code...
}
else
{
    code...
}
```

The order of the execution is always linear, starting from the first Boolean expression moving forward towards the last `else` statement. If any of the Boolean expressions is True, then the rest of the Boolean expressions is discarded. Please note that the last `else` statement is optional. You may decide to include/exclude it based on your programming logic.

Now, we will update our existing code to display the letter grade instead of the Success/Failure messages. We will basically use the `else if` statements to check for multiple conditions. Please examine the code provided in Figure 19 and change your code accordingly.

```
if(grade >= 90)
{
    MessageBox.Show("A");
}else if (grade >= 80)
{
    MessageBox.Show("B");
}
else if (grade >= 70)
{
    MessageBox.Show("C");
}
else if (grade >= 60)
{
    MessageBox.Show("D");
}
else
{
    MessageBox.Show("F");
}
```

Figure 19. Adding the else if statements.

Let's run the application and check if the code runs correctly. Please press F5. You can do several trials; the code should work fine (see Figure 20).

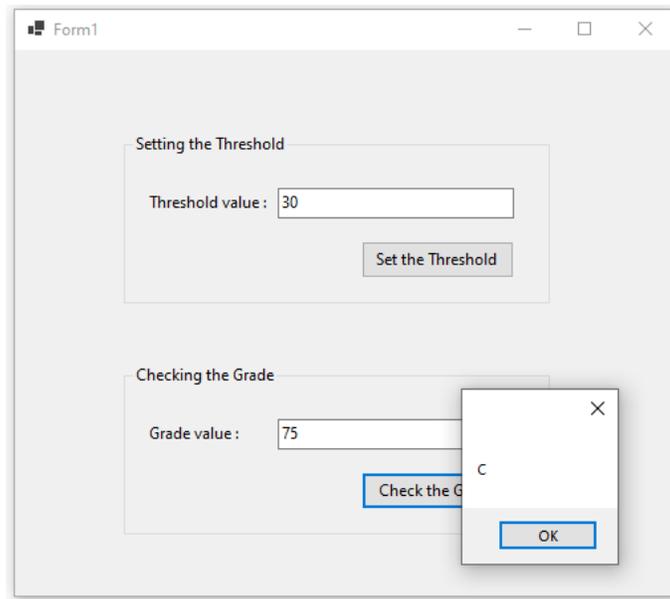


Figure 20. Testing the application.

Right now, the code works fine but it could cause some problems if we have ordered the conditional statements differently. As an example, we will do a slight change in the order of the `else if` statements and check how it effects the output. We will move the `else if (grade >= 70)` code block before `else if (grade >= 80)`. After this change, your code should like the code shown in Figure 21.

```
if (grade >= 90)
{
    MessageBox.Show("A");
}
else if (grade >= 70)
{
    MessageBox.Show("C");
}
else if (grade >= 80)
{
    MessageBox.Show("B");
}
else if (grade >= 60)
{
    MessageBox.Show("D");
}
else
{
    MessageBox.Show("F");
}
```

Figure 21. Changing the order of the `else if` statements.

First enter 30 for the threshold value, although we are not using it right now. If you do not provide a value for the threshold field, you should receive an error, do you guess why? This is because we try to convert it to an `int` value. A blank value cannot be casted to `int`, and an exception would be thrown.

Next please enter 85 for the grade field and press the *Check the Score* button. You should see **C** in the output window instead of **B**, which should have been printed (see Figure 22).

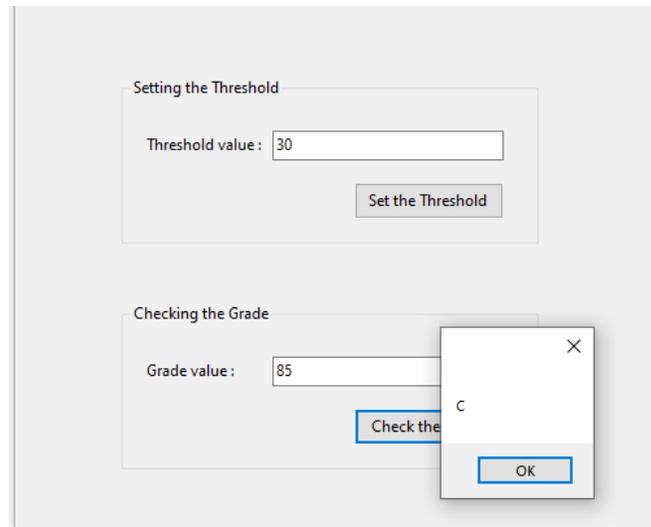


Figure 22. Testing the application after switching the else if statements.

This logic error was because the program exists from the *if-else* block, once any of the conditions holds. Let's go through the code line by line to understand what has happened. The first Boolean expression ($\text{grade} \geq 90$) will be *false* and therefore any code between the opening and ending braces, i.e., `{ MessageBox.Show("A") }`, will be discarded. Then the next Boolean expression will be evaluated: ($\text{grade} \geq 70$), which is *true* since 85 is greater than 70. Therefore, `MessageBox.Show("C")` will be executed, and C will be printed. The program will exit from the *if* code block to continue with the next line of code. That is, the *intended* Boolean expression ($\text{grade} \geq 80$) will never get executed for grade values greater than 80.

```
if (grade >= 90)           //false
{
    MessageBox.Show("A");//skipped
}
else if (grade >= 70)      //true since 85 is greater than 70
{
    MessageBox.Show("C");//executed, and C is printed.
}
else if (grade >= 80)      //starting from here, the rest is not executed.
{
    MessageBox.Show("B");
}
else if (grade >= 60)
{
    MessageBox.Show("D");
}
else
{
    MessageBox.Show("F");
}
```

Figure 23. The program logic explained within the code.

To better test how if-else statements work, we will slightly change our code. Please change the code as shown in Figure 24.

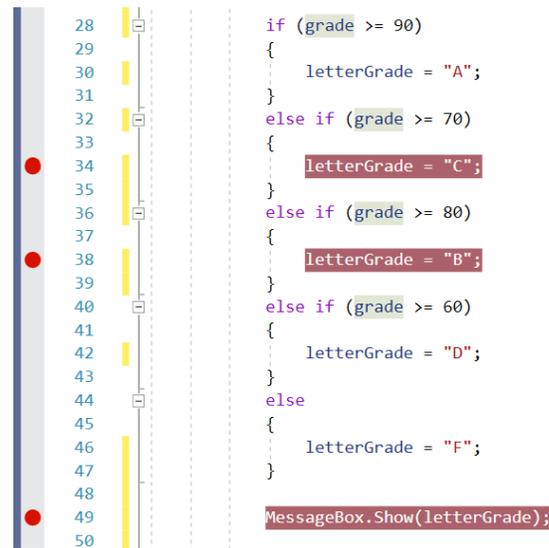


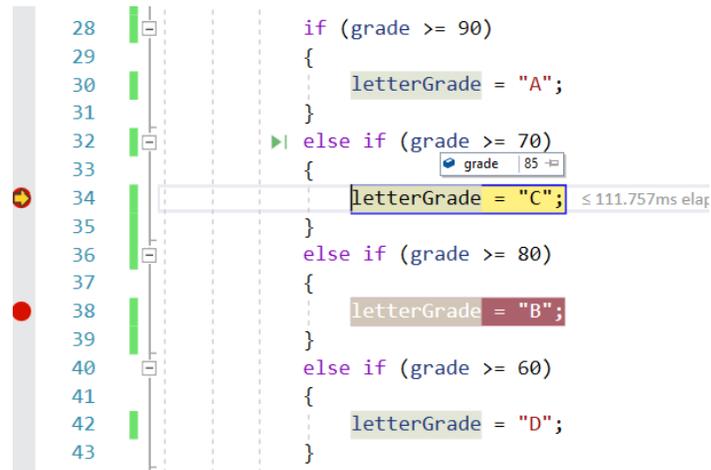
Figure 24. Creating the letterGrade variable and adding breakpoints..

The main change is that we declared a new string variable called `letterGrade`, and in the code block following each conditional statement, we assigned the intended letter grade value to this variable. Then, outside the if-else block, we print the `letterGrade` variable using `MessageBox.Show()` method.

Now, we will add breakpoints at the lines 34, 38, and 49 as shown in Figure 24 to see how the program process the whole `if` code block. To do so, we will run our application with debugging mode enabled by pressing F5. Once the form appears on the screen, please enter the following threshold and grade values as shown in Figure 25 and press the *Check the Grade* button.

Figure 25. Entering the form values.

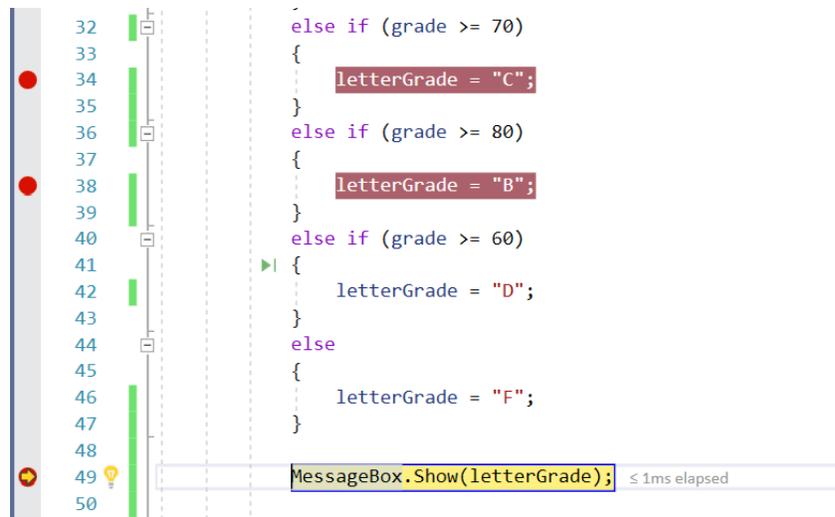
The program should pause at the first breakpoint since the `grade >= 70` is true (see Figure 26).



```
28     if (grade >= 90)
29     {
30         letterGrade = "A";
31     }
32     else if (grade >= 70)
33     {
34         letterGrade = "C";
35     }
36     else if (grade >= 80)
37     {
38         letterGrade = "B";
39     }
40     else if (grade >= 60)
41     {
42         letterGrade = "D";
43     }
```

Figure 26. The program pauses at the first breakpoint.

We will click on Continue (at the top bar) to continue with the execution of the program. Now, the program should stop at the third breakpoint (line 49). That means, the second breakpoint at line 38 was not activated since the program skipped the rest of the if code block, and it jumped to the code (line 49) that follows just after the if code block (see Figure 27). This code will assign the "C" value to the `letterGrade` variable. If you click on Continue, the "C" should be displayed in the popup window.



```
32     else if (grade >= 70)
33     {
34         letterGrade = "C";
35     }
36     else if (grade >= 80)
37     {
38         letterGrade = "B";
39     }
40     else if (grade >= 60)
41     {
42         letterGrade = "D";
43     }
44     else
45     {
46         letterGrade = "F";
47     }
48
49     MessageBox.Show(letterGrade);
50
```

Figure 27. The program pauses at the third breakpoint.

5) Creating compound Boolean expressions with logical operators

Sometimes we need to combine two or more Boolean expressions to check for a condition. For example, to check if a student achieved a quiz grade above threshold and under certain number of attempts, we need to create a compound Boolean expression. We can use logical operators to create such complex

Boolean expressions by combining two or more subexpressions. Table 1 provides the list of logical operators along with several examples.

Table 1. Logical operators

| Operator | Description | Examples |
|----------|--|--|
| && | Logical AND operator: Returns true only if both subexpressions are true | <i>//if a is greater than 15 AND b is equal to 20</i> <code>a > 15 && b == 20</code> <i>//if quizScore is higher than 50 AND quizAttempts is less than 4</i> <code>quizScore >50 && quizAttempts <= 3</code> |
| | Logical OR operator: Returns true if one of the subexpressions is true. | <i>//if x is less than 15 OR x is higher than 50</i> <code>x < 15 x > 50</code> <i>//if quizScore1 is higher than 60 OR quizScore2 is higher than 60</i> <code>quizScore1 > 60 quizScore2 > 60</code> |
| ! | Logical NOT operator: Reverses the truth of a Boolean expression. | <i>If a > 50 is NOT true</i> <code>!(a > 50)</code> <i>//if a is NOT between 50 and 60</i> <code>!(a > 50 && a <= 60)</code> |

As you may remember, our application outputs incorrect letter grade for values greater than 80. We can fix this by creating a compound Boolean expression that checks if the grade is higher than 70 and lower than 80. We will update the `else if` statement at line 32 as a compound Boolean expression using the `&&` operator (see Figure 28) to check if the grade value falls between 70 and 80.

```

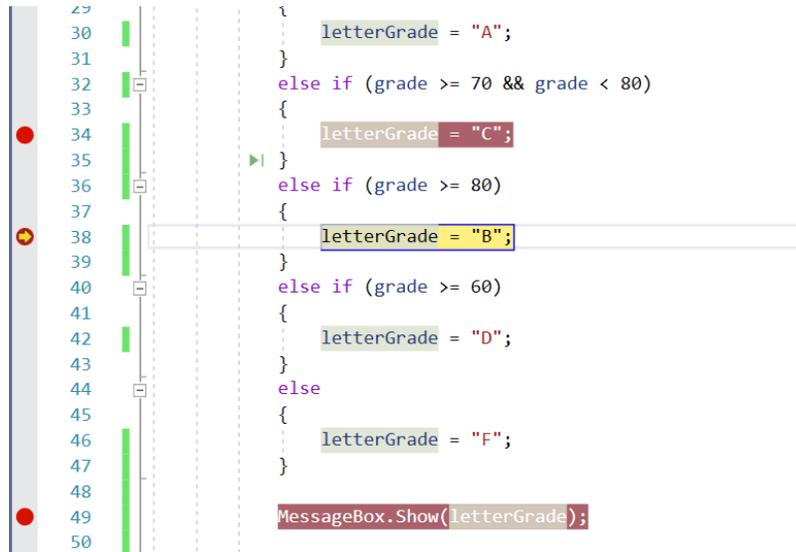
28 | if (grade >= 90)
29 | {
30 |     letterGrade = "A";
31 | }
32 | else if (grade >= 70 && grade < 80)
33 | {
34 |     letterGrade = "C";
35 | }
36 | else if (grade >= 80)
37 | {
38 |     letterGrade = "B";
39 | }
40 | else if (grade >= 60)
41 | {
42 |     letterGrade = "D";
43 | }
44 | else
45 | {
46 |     letterGrade = "F";
47 | }

```

Figure 28. Adding a compound Boolean expression.

We will run our application again (press F5). Please enter 30 for the threshold and 85 for the grade fields, and then click on the *Check the Grade* button. This time the program should pause at the second

breakpoint (see Figure 29), where “B” is assigned to the letterGrade. That means our if-else-if logic is working fine this time.



```
30 letterGrade = "A";
31 }
32 else if (grade >= 70 && grade < 80)
33 {
34 letterGrade = "C";
35 }
36 else if (grade >= 80)
37 {
38 letterGrade = "B";
39 }
40 else if (grade >= 60)
41 {
42 letterGrade = "D";
43 }
44 else
45 {
46 letterGrade = "F";
47 }
48
49 MessageBox.Show(letterGrade);
50
```

Figure 29. The program pauses at the second breakpoint.

If you continue running the application, “B” will be displayed in the popup window as seen in Figure 30.

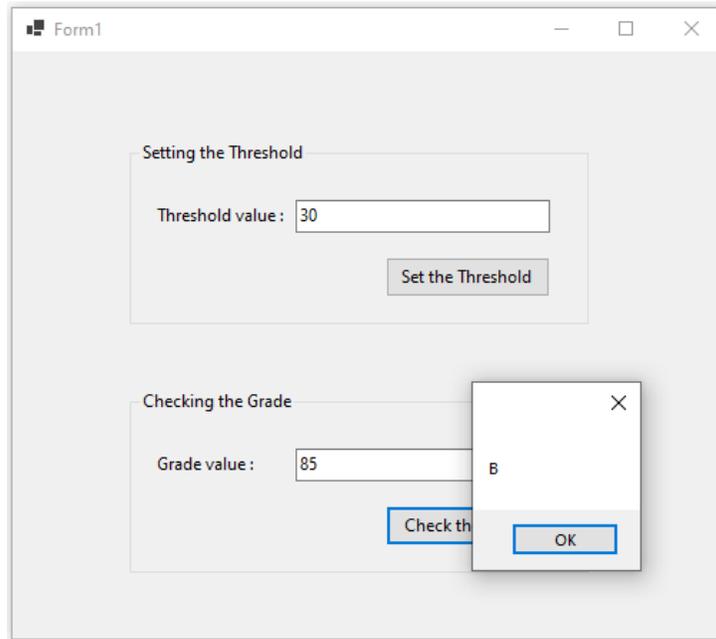


Figure 30. The program outputs “B”.

Actually, it makes more sense in all cases to define intervals and check if the grade falls into any of them for deciding which letter grade to assign. We will repeat what we have already done for all else-if condition statements. Please update your code as shown in Figure 31.

```

if (grade >= 90)
{
    letterGrade = "A";
}
else if (grade >= 80 && grade < 90)
{
    letterGrade = "B";
}
else if (grade >= 70 && grade < 80)
{
    letterGrade = "C";
}
else if (grade >= 60 && grade < 70)
{
    letterGrade = "D";
}
else
{
    letterGrade = "F";
}

```

Figure 31. Creating compound expressions for all else if statements.

6) Further example with nested decision structures

As we mentioned before, you can place a decision structure inside another one to create more complex decision mechanisms. For example, in our application, we may check first if a threshold is provided. With a valid threshold value, we can decide if students fail or pass the course if their grade is above or below the threshold, without calculating the grade letter. If no threshold is provided, then we can use the existing `if-else-if` statements to find out the correct letter grade and print it (as we did already).

To begin with this change, please first remove the `threshold` variable from `btn_checkGrade_Click`. Then, define `threshold` as a class variable and initialize it with 0, as shown in Figure 32.

```

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

    private int threshold = 0;

    private void btn_checkGrade_Click(object sender, EventArgs e)
    {
        int grade = int.Parse(txt_grade.Text);

        string letterGrade;
    }
}

```

Figure 32. Defining threshold as class variable

Now, we will create a click event handler for the Set the Threshold button, and inside the handler method, we will set the value of the threshold to what is typed inside the `txt_threshold` textbox (see Figure 31).

```

private void btn_setThreshold_Click(object sender, EventArgs e)
{
    threshold = int.Parse(txt_threshold.Text);
}

```

Figure 33. Creating the click event handler and setting the value of threshold.

So far, what we have done is to read the threshold value provided by the user into a class-level variable: `threshold`. This variable will be accessible from all methods (e.g., different click event handlers) within the same class (Form1.cs). If no value is set, the value of `threshold` is 0.

Now, we will move to the `btn_checkGrade_Click` click event handler. We will create a nested decision structure. Do not delete what you have already have to determine the correct grade letter. We will use it soon. First, create a new if-else condition that checks if the threshold is 0 or not, as shown in Figure 32.

If threshold is equal to 0, we will decide on the correct grade letter (for which, we already have the code). If not, we will print fail (`grade < threshold`) or pass (`threshold < grade`).

```

if(threshold == 0)
{
    //Identify the grade letter
}
else
{
    //Print failr or pass
}

```

Figure 34. Checking if threshold is 0 or not.

Please move your existing if-else-if statements inside the if code block. Your code should look like Figure 33.

```

if(threshold == 0)
{
    if (grade >= 90)
    {
        letterGrade = "A";
    }
    else if (grade >= 80 && grade < 90)
    {
        letterGrade = "B";
    }
    else if (grade >= 70 && grade < 80)
    {
        letterGrade = "C";
    }
    else if (grade >= 60 && grade < 70)
    {
        letterGrade = "D";
    }
    else
    {
        letterGrade = "F";
    }
    MessageBox.Show(letterGrade);
}

```

Figure 35. Computing the letter grade.

Now, write the necessary code for the else part (i.e., threshold is not 0) (see Figure 32) to print Fail or Pass messages, as shown in Figure 34.

```
else
{
    if (threshold > grade)
        MessageBox.Show("Fail");
    else
        MessageBox.Show("Pass");
}
```

Figure 36. Printing the Fail or Pass messages.

Please test your application. If you never set the threshold value, a letter grade should be displayed (see Figure 35). If you set a threshold value, then Fail or Pass messages should be displayed depending on the value of the grade provided (see Figure 36).

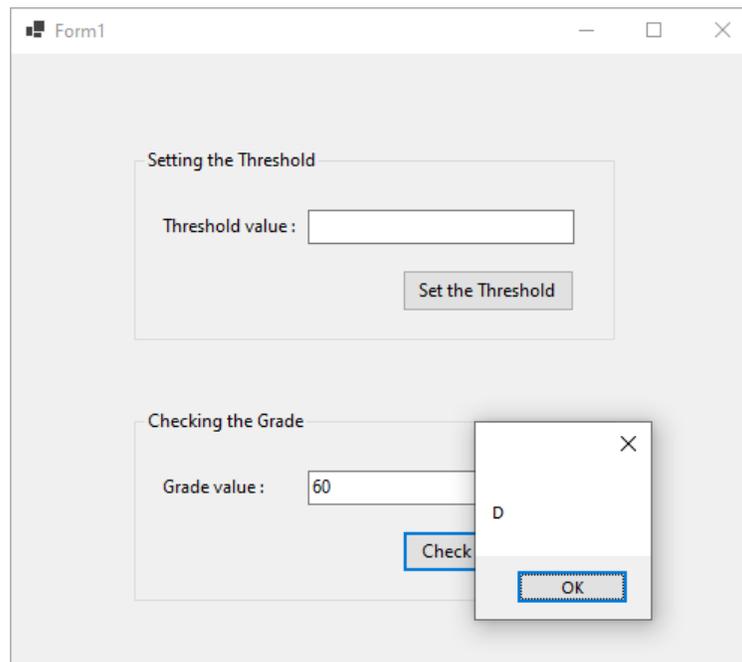


Figure 37. No threshold value is set.

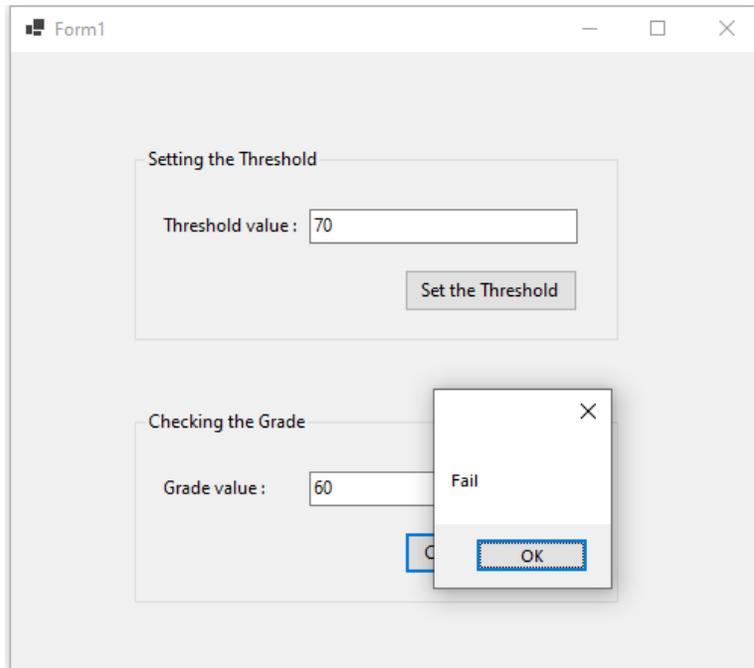


Figure 38. Threshold value is set.