

Module 6 – Loops, arrays, and lists.

One of the fundamental concepts in programming is loops. Loops enable us to repeatedly run a portion of code as long as a specific condition is met. Repetitive tasks are quite common (e.g., sending an email to every individual in a list of 100 students) and loops help us save time to handle these tasks with maximum efficiency. For example, when you see a list of products in a website or application, that means probably some looping has been involved in the coding to iterate through these products and display them properly.

We will cover 4 types of loops available in C#. To continue with the rest of this chapter, please create a new project called Module6_1.

1) While loop

As its name suggests, **while** loop repeats a portion of code *while* the Boolean expression specified is **true**. We will write an example code to test how while loop functions. Please add the following code inside the Form1_Load method (see Figure 1).

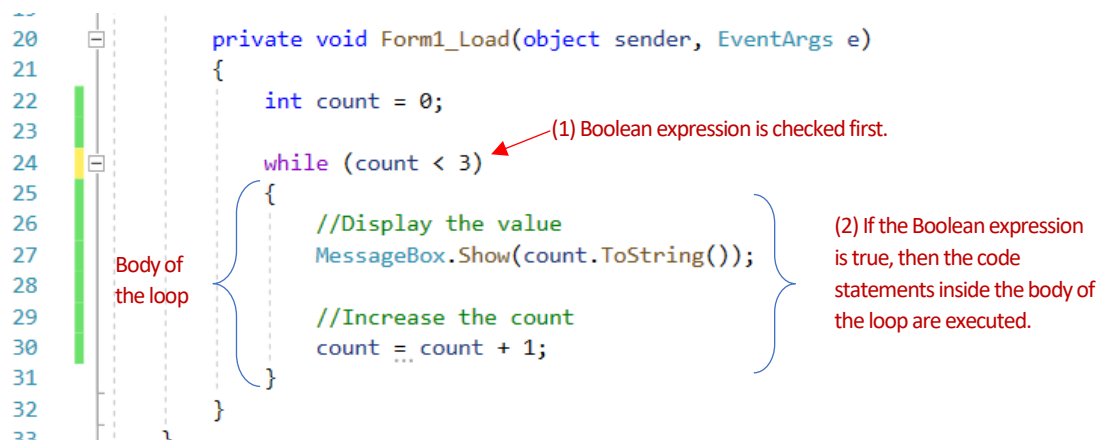


Figure 1. While loop example.

In each iteration, **while** loop first evaluates the Boolean expression, and then executes the statements inside the body of the loop if the Boolean expression is true. If it is false, then the program exits from the loop and goes to the next line after the loop code block.

Considering the example given in Figure 1, in the first iteration the value of `count` will be 0. Since it is less than 3, the Boolean expression will be evaluated to true. Therefore, the program will continue with executing the statements inside the body of the loop. It will first run the code at line 27 to display the current value of `count`, which is 0, and then it will run the code at line 30 to increase the value of `count` by one (thus, its new value will be 1). The first iteration will be completed, and the program will continue with line 24 to start the second iteration.

The second iteration will be the same as the first one, except that the value of `count` will be 1. Since it is still less than 3 (that is the Boolean expression is evaluated to true), the program will again execute the

body of the loop. After printing its value, the program will increase the value of `count` by one and the second iteration will be completed.

In the third iteration the condition will be still met since the value of `count`, 2, is still less than 3. At the end of the third iteration the value of `count` will be 3.

In the third iteration, this time the Boolean expression will be false since the current value of `count`, which is 3, is NOT less than 3. Therefore, the loop will be terminated, and the program will continue with the next statement after the loop.

Please note that defining a counter variable (which is `count` in the example above) is essential in many loops. Not only it helps to identify the number of iterations, but also it prevents infinite loops. Infinite loops occur when the conditional statement is always met (thus Boolean expression is evaluated as true). For example, in the code above, not incrementing the count variable would result in an infinite loop since `count` would be always less than 3.

2) For loop

`for` loop is a more specified version of `while` loop. You may remember that in `while` loop, we defined the `count` variable before the loop, and increased its value inside the loop body. Differently in `for` loops, these tasks are performed inside a *for header*.

The code shown in Figure 2 uses `for` loop to perform the same operation that we did with `while` loop.

```
24 for (int count = 0; count < 3; count++) ← for header
25 {
26     //Display the value
27     MessageBox.Show(count.ToString());
28 }
```

Figure 2. For loop example.

As shown in Figure 3, for loop header is composed of three expressions separated by a semicolon.

```
for (int count = 0; count < 3; count++)
```

Figure 3. Components of the for header.

1. **Initialization expression:** This expression initializes a counter variable with an initial value. If the `counter` variable were to be defined beforehand, it would still need to be assigned a value inside the for header. This expression is evaluated **ONLY ONCE before the first iteration** starts.
2. **Boolean expression:** This expression controls if the statements inside the *body* of the `for` loop should be executed or not. This expression is evaluated at the beginning of each iteration, and as long as it is true, the body of the loop gets executed.
3. **Update expression:** This expression updates the value of the `counter` variable (generally, increments its value), and is executed at the **end** of each iteration.

Lets' go through how the `for` loop presented in Figure 2 will be executed in runtime. For the first time, `for` header will be executed, in which `count` variable will be declared and set to 0 and it will be checked if its value is less than 3. The update expression incrementing its value by 1 will NOT be executed initially. Since 0 is less than 3, the body of the `for` loop will be executed and the `count` value, 0, will be displayed in a popup window. At the end of the iteration, the update expression inside the `for` header will be executed and the value of `count` will be incremented by 1. Please not that, here we used the `++` operator to increment the value by one (`count++`), which is equivalent to the `count = count + 1` expression.

At the beginning of the second iteration, only the Boolean expression will be executed inside the `for` header, which will be true (since 1 is less than 3), and then the body of the loop will be executed, which will print the current `count` value, 1. At the end of the iteration, the update expression inside the `for` header will be executed and the value of `count` will increase to 2.

The third iteration will be executed in the same manner as the second iteration. The value of the `count` variable, which is 2, will be printed as the output. At the end of the third iteration, `count` will be set to 3. In the fourth iteration, the Boolean expression will be false (since 3 is not less than 3), and the body of the loop will not be executed. The program will exit from the loop code block and continue with the next line.

3) Do-while loop

This loop is very similar to `while` loop, except that the conditional `while` statement comes at the end after the body of the loop is executed first. That is, `do-while` loops are *post-test* whereas `for` and `while` loop are called *pre-test* loops. The Figure 4 shows how to use `do-while` loop to print the numbers from 0 up to 3.

```
int count = 0;

do
{
    MessageBox.Show(count.ToString());
} while (count < 3);
```

Body of the loop

while header

Figure 4. Do-while loop example.

The code will produce the same output as the `while` and `for` loops. The only difference is that, if the `count` variable was set to 3, the body of the loop would be executed and the value of 3 would be printed out, which would not be possible with the other loops.

Although it is very rare to have a specific case where you are obliged to use `do-while` loop, you may need it if you have to run the body of the loop at least once before the Boolean expression is checked in the `while` header.

4) Foreach loop

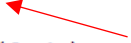
The last type of loops is the `foreach` loop, which is very useful and efficient in iterating a list of variables or objects. The number of iterations in `foreach` loop is determined by the length of the list. If the list has 10 items, there will be 10 iterations. A counter variable is NOT necessary.

Figure 5 shows an example code of `foreach` loop, where it iterates through a `string array`, called `fruits`, holding 4 items. Soon, we will learn about arrays. For now, we will focus on the loop.

```
string[] fruits = new string[3];

fruits[0] = "Apple";
fruits[1] = "Mango";
fruits[2] = "Kiwi";

foreach(string fruit in fruits)
{
    MessageBox.Show(fruit);
}
```



In the 1st iteration `fruit` is set to `fruits[0]`,
In the 2nd iteration `fruit` is set to `fruits[1]`,
In the 3rd iteration `fruit` is set to `fruits[2]`.

Figure 5. For each loop example.

Starting from the first item, `fruits[0]`, which is stored at the 0 index, `foreach` loop iterates through the `fruits` array until the last item (which is `fruits[2]`) in the list is reached. The body of the loop is executed at each iteration. Although this `foreach` loop can be translated into `for` loop (see Figure 6), I recommend you use `foreach` loop when iterating through a list of values.

```
fruits[0] = "Apple";
fruits[1] = "Mango";
fruits[2] = "Kiwi";

for(int index=0; index < fruits.Length; index++)
{
    MessageBox.Show(fruits[index]);
}
```

Figure 6. Using `for` loop to iterate through the fruit array.

In the next sections, we will cover arrays and lists and further practice how we can use loops to iterate through list objects.

5) Array objects

So far we have used variables to store values (or data) in the memory. One limitation of variables is that they can hold only one value at a time. Each time you assign a value to a variable, the previous value is replaced and cannot be recovered. Often we need to store and process a list of values. Imagine that our program needs to calculate the final grade for 100 students based on their exam scores. It is impractical to create variables one by one for each student and perform calculations over and over again.

One option to store a set of values is **array**. Arrays can hold a group of values that have the same data type. For example, we can have a `string` array or an `int` array, but we *cannot* define an array that holds a mix of `string` and `int` values.

Figure 7 provides an example code about how to declare arrays. In the first example, an empty `string` array with the size of 3 (i.e., it can hold up to 3 values) is created using the `new` keyword, and the first element is set to "Apple". Please note that in C# arrays are **zero-based**. This is why the index of the first element is 0. The second line declares an empty `int` array with the size of 2. Since, it can hold two values, the first element will be at index 0 while the second element will be stored at index 1. The value of the second element is set to 15.

```
string[] fruits = new string[3];  
fruits[0] = "Apple";  
  
int[] numbers = new int[2];  
numbers[1] = 15;
```

Figure 7. Declaring string and int type arrays.

Arrays are **reference** type objects. When you define an array as in Figure 7, it is created and stored in the memory and a reference to this array object is returned (instead of the object itself). For example, `numbers` variable holds the *reference* (address) value to access the array object in the memory. This is different than the **value** type variables we have used so far.

6) Value type vs reference type variables

In C#, variables are divided into two categories based on their data types: **value** type and **reference** type variables. The `int`, `string`, `double`, `decimal`, and `float` variables we have used so far in this course are **value** type variables. When a value type variable is declared, a part of memory (e.g., 4 byte if it is an `int` variable) is allocated to store the value of the variable. For example, if you define an `int` variable with a value of 500, a specific portion in the memory will be located to store this value as illustrated in Figure 8.

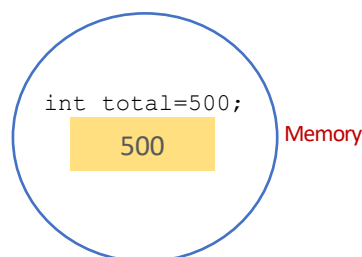


Figure 8. Location assigned in the memory for total variable.

On the other hand, **reference** type variables work totally different. Reference type variables hold a reference (address) to the object stored in the memory. In other words, they do not store the object itself, but the address of the object stored in the memory. Therefore, when assigning a reference variable to another one, you will not copy the data but the reference which refers to the same location in the memory. For example, arrays are reference types, and when they are created in the memory, a reference to their location in the memory is returned and stored in the reference variable as illustrated in Figure 9.

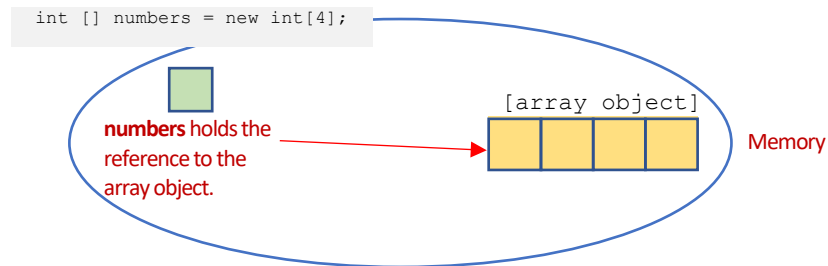


Figure 9. Location assigned in the memory for the array object.

7) Indexing in arrays and the length property

As you might notice from the examples above, a specific element inside an array is accessed and updated through the index indicated inside square brackets. As shown in Figure 10, please declare the `numbers` array with size of 4. Then, assign a value to each of the element in the array as indicated in Figure 10.

```
int[] numbers = new int[4];
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
numbers[3] = 40;
```

Figure 10. Definition and initialization of the `numbers` array.

Figure 11 shows the elements stored inside the number array.

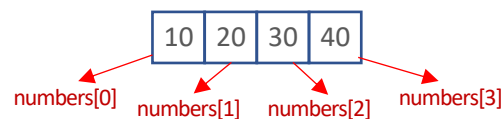


Figure 11. The elements of the `number` array.

Since arrays follow zero-based indexing, the last element in an array is always accessible through the index equal to the length of the array **minus 1**. For example, the last element of the `numbers` array is at index 3, which is one less than its size. Sometimes, you may not remember the size (or length) of an array. You can use the **Length** property to know how many elements an array can hold. The following code in Figure 12, prints the last element of the `numbers` array.

```
MessageBox.Show(numbers[numbers.Length - 1].ToString());
```

Figure 12. Printing the last element of `numbers` array.

You can actually initialize an array with a set of values if you already know what initial values the array should have. The syntax for initializing the `numbers` array with some values is shown in Figure 13.

```
int[] numbers = { 10, 20, 30, 40 };
```

Figure 13. Initializing the `numbers` array with a set of values.

When initializing an array with a set of values, you do not need to specify either the new keyword or the size of the array. The size of the array is automatically set to the number of items in the list of values assigned to the array. In the example above, the size of the array will be 4.

8) Duplicating and comparing arrays

Copying an array is not very straightforward. Since an array is an object and we define a variable to store reference to that object in the memory. When we assign this variable to another, we actually copy the reference, not the array object itself.

In the code shown in Figure 14, first an array object which holds 4 values is created and then the reference to this array object is set to `luckyNumbers`. When, we assign the value of the `luckyNumbers` variable to `backupNumbers`, we actually pass the reference value not the array object. Therefore, if we perform any changes on `backupNumbers`, we actually manipulate the same array object in the memory, as illustrated in the figure.

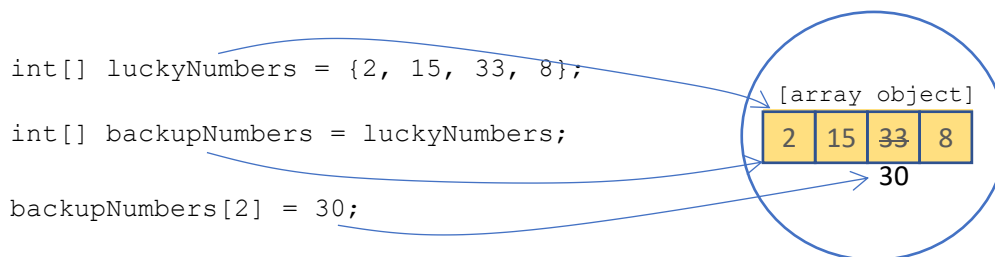


Figure 14. `luckyNumbers` and `backupNumbers` share the same reference to the array object.

To duplicate an array, first you need to create a new empty array object in the memory. Then, you should iterate through the original array and copy its individual elements one by one to the new array at each iteration. Obviously, we need to use a loop to do this. Figure 15 shows how to duplicate the `luckyNumbers` array using `for` loop.

```
int[] luckyNumbers = { 2, 15, 33, 8 };
int[] backupNumbers = new int[luckyNumbers.Length];

for(int index=0; index < backupNumbers.Length; index++)
{
    backupNumbers[index] = luckyNumbers[index];
}
```

Figure 15. Using `for` loop to duplicate the `luckyNumbers` array.

How about checking if two arrays are identical? You may remember that we use the `==` operator to check if two variables hold the same value or not. Unfortunately, this won't work for arrays which are reference types. For example, the code below will compare the reference values of `luckyNumbers` and `backupNumbers` arrays, not their contents, and the `if` conditional statement will always return `false`.

```
int[] luckyNumbers = {2, 15, 33, 8};
```

```

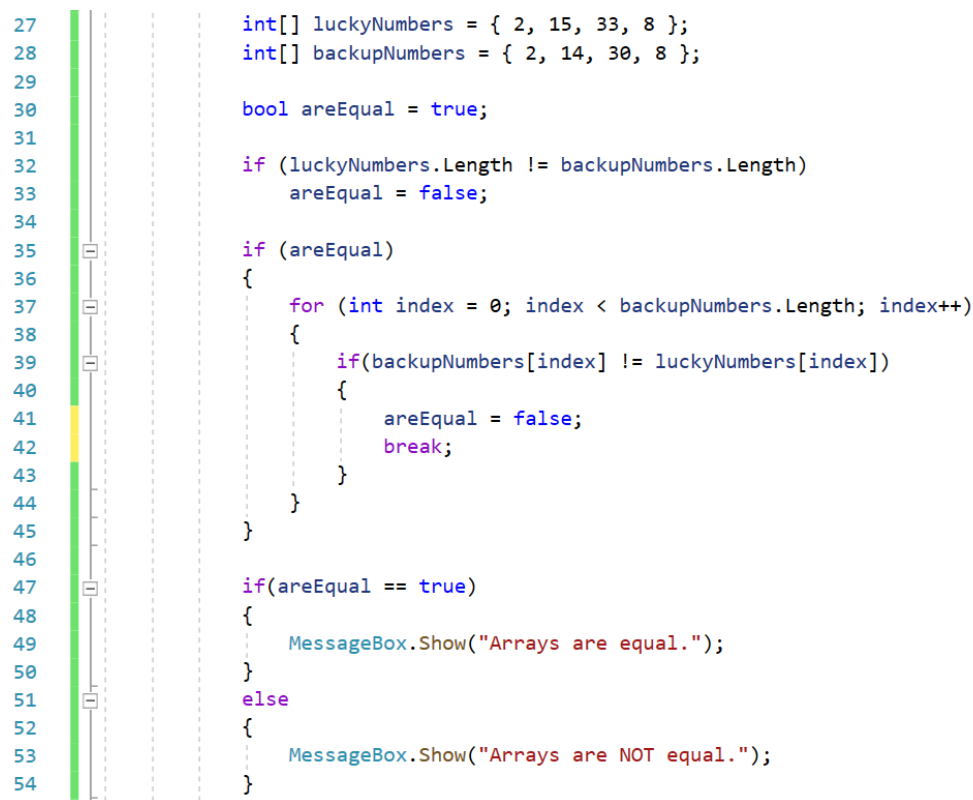
int[] backupNumbers = {2, 14, 30, 8};

if(luckyNumbers == backupNumbers){
    MessageBox.Show("Equal!");
}

```

To check if the two arrays are equal, we need to first check if they have the same number of elements. If this is true, then we need to iterate through the array to check if the elements at the same index are the equal or not. If there are any mismatches, we should return false.

The code provided in Figure 16 uses `for` loop to compare two arrays. After defining the arrays, at line 30 we define a `bool` variable, `areEqual`, which will be used throughout the code to signal if the arrays are equal or not. At lines 32-33 we check if the arrays have the same length, if not, we set `areEqual` to false. Then, if `areEqual` is true (line 35), that is the arrays have the same size, we use `for` loop to iterate through the arrays. At each iteration, we check if the values stored in the same index of two different arrays are equal (line 39), and if not, we set `areEqual` to false (line 41).



```

27 int[] luckyNumbers = { 2, 15, 33, 8 };
28 int[] backupNumbers = { 2, 14, 30, 8 };
29
30 bool areEqual = true;
31
32 if (luckyNumbers.Length != backupNumbers.Length)
33     areEqual = false;
34
35 if (areEqual)
36 {
37     for (int index = 0; index < backupNumbers.Length; index++)
38     {
39         if(backupNumbers[index] != luckyNumbers[index])
40         {
41             areEqual = false;
42             break;
43         }
44     }
45 }
46
47 if(areEqual == true)
48 {
49     MessageBox.Show("Arrays are equal.");
50 }
51 else
52 {
53     MessageBox.Show("Arrays are NOT equal.");
54 }

```

Figure 16. Using `for` loop to compare arrays.

Please note the `break` statement at line 42. `break` will exit the loop immediately without running any further iterations and the program will continue with the next statement after the loop. Here, we use `break` because we do not need to iterate through the whole array once we find any mismatch. Between lines 47-54, we use `if-else` to print a message whether the arrays are equal or not.

The code in Figure 17 shows how to use `while` loop to compare the arrays. With `while` loop, the code is cleaner and simpler since we do not have to use `if` or `break` statements. It seems that `while` loop is a better choice for this task. Sometimes, it is important to try out different options to figure out which one is more efficient in terms of performance.

```
int[] luckyNumbers = { 2, 15, 33, 8 };
int[] backupNumbers = { 2, 14, 30, 8 };
int index = 0;
bool areEqual = true;

if (luckyNumbers.Length != backupNumbers.Length)
    areEqual = false;

while (areEqual == true && index < backupNumbers.Length)
{
    if (backupNumbers[index] != luckyNumbers[index])
    {
        areEqual = false;
    }
    index++;
}

if (areEqual == true)
{
    MessageBox.Show("Arrays are equal.");
}
else
{
    MessageBox.Show("Arrays are NOT equal.");
}
```

Figure 17. Using `while` loop to compare arrays.

9) The list collection

As a strong alternative to arrays, C# provides the `List` collection. `List` is actually a .Net framework class, and any objects you create from this class will be a `List` type. A `List` object has many advantages over an `array` since it automatically expands or shrinks as new items are added or removed from it. `List` objects can hold any value types or objects. For example, the following lines of code create two types of `List` objects, referenced by `studentEmailList` and `studentGradeList`, one holds strings while the other one holds integers.

```
List<string> studentEmailList = new List<string>();

List<int> studentGradeList = new List<int>();
```

Please note that, following the `List` keyword inside the angle brackets, we define the data types that the `List` will hold, such as `<int>` or `<string>`.

You can also set some initial values to a `List` object when you first time declare it. The following code shows an example:

```
List<int> studentGradeList = new List<string>() {85, 90, 85};
```

To add items to a `List`, you can use its **Add** method. For example, the following code create a list object to store some names, and then we add three names to the `List`.

```
List<string> names = new List<string>();  
names.Add("Kerim");  
names.Add("Gustavo");  
names.Add("Florian");
```

The items in a `List` are stored at specific indexes as in arrays. The first item ("Kerim") is stored at index 0, the second item ("Gustavo") is stored at index 1, and the third item ("Florian") is stored at index 2. As we did in arrays, you can use these indexes to access a specific value in a `List`. For example, `names[0]` will return the value of "Kerim".

The following code uses a `for` loop to print each item in the `names` list:

```
List<string> names = new List<string>();  
  
for (int index = 0; index < names.Count; index++)  
{  
    MessageBox.Show(names[index]);  
}
```

Please note that, in the `for` statement, we need to ensure that the `index` value is within the boundaries of the `List` size by checking that `index` is less than the size of the `List`. For that purpose, we used the **Count** property, which returns the number of items that a `List` holds.

Different from the **Add** method, which always appends the new item to the last index of the `List`, we can use the **Insert** method to add an item at a specific index. In the following example, we insert "David" as the first item in the `List` using index 0.

```
names.Insert("David", 0) //{ "David", "Kerim", "Gustavo", "Florian" }
```

To remove an item from a `List` object, we can use two methods: `RemoveAt()`, which removes an item at a specific index, or `Remove()` which removes the item that matches the provided string. The following code provides examples for these methods:

```
names.RemoveAt(0) //{ "Kerim", "Gustavo", "Florian" }  
names.Remove("Florian") //{ "Kerim", "Gustavo" }
```

The `Remove` method returns **true** if the indicated value exists in the `List` and returns false otherwise. For example, the code below will remove "Florian" if exists in the `names` list, otherwise, it will print a message to inform the user that the item was not found.

```

if(!names.Remove("Florian"))
{
    MessageBox.Show("This item was not found.");
}

```

We can check if a `List` contains a specific item by using the `IndexOf()` method. This method returns -1 if the item was not found in the `List`. Otherwise, it returns the index of the item in the `List`. The following code searches for "Kerim" in the `names` list and prints its index if found. The user is informed if the item is not in the list.

```

int position = names.IndexOf("Kerim");

if(position == -1){
    MessageBox.Show("This item was not found.");
}
else
{
    MessageBox.Show("The item was found at index " +
        position.ToString());
}

```

10) Programming exercise

We will end this chapter with a programming exercise in which we will develop an application that prints student names in two sections of a course inside a list box.

First, we will build the interface. For this, please switch to the design view and add a listbox control and under it, add a button control, as shown in Figure 18. Please name these controls properly.

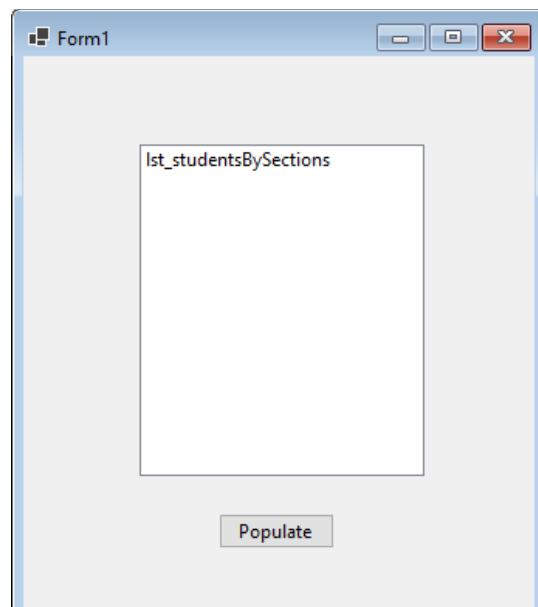


Figure 18. The design of the form.

Our goal is, when clicked on the button, to populate the list box with student names divided by the sections as shown in Figure 19.

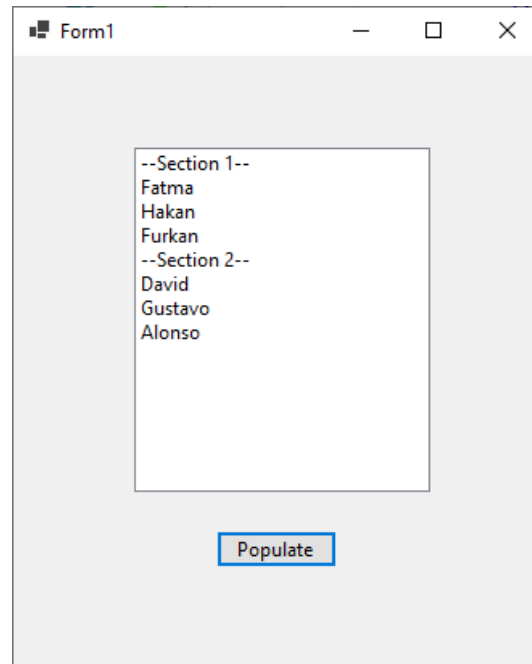


Figure 19. Sample output of the application.

Therefore, the code should go inside the click event handler of the button control. Create the event handler as shown in Figure 20.

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

Figure 20. Empty click event handler.

Next, we will generate our data. Our data will be consisted of two `string` arrays initialized with the following sets of values:

```
string[] section1 = { "Fatma", "Hakan", "Furkan" };
string[] section2 = { "David", "Gustavo", "Alonso" };
```

Then, we will create `List` object named `sections` that can hold `string[]` type. Next, we will add the `section1` and `section2` arrays to the `List` object:

```
List<string[]> sections = new List<string[]>();
sections.Add(section1);
sections.Add(section2);
```

The click event handler should have the code shown in Figure 21.

```
private void btn_submit_Click(object sender, EventArgs e)
{
    string[] section1 = { "Fatma", "Hakan", "Furkan" };
    string[] section2 = { "David", "Gustavo", "Alonso" };

    List<string[]> sections = new List<string[]>();
    sections.Add(section1);
    sections.Add(section2);
}
```

Figure 21. Click event handler with data generation code.

We need to iterate through the `list` object, and for each section array stored in this list, we need to iterate through the student names. That is, we need to implement **nested** loops: *outer* loop will go through the content of the `sections` list, while the *inner* loop will go through the elements inside the `section1` and `section2` arrays.

Let's use `for` loop for the outer loop which will iterate through the `sections` list as shown below:

```
for (int index = 0; index < sections.Count; index++)
{
    int sectionNo = index + 1;
    lst_studentsBySections.Items.Add("--Section " +
                                    sectionNo.ToString() + "--");
}
```

At each iteration, we need to add the section number which will be one more than the index value: that is, 1 if the index is 0, and 2 if the index is 1. We use the **Items.Add** method of the listbox to add new items to the list.

If you run your application and click the button, you should obtain the following output shown in Figure 22.

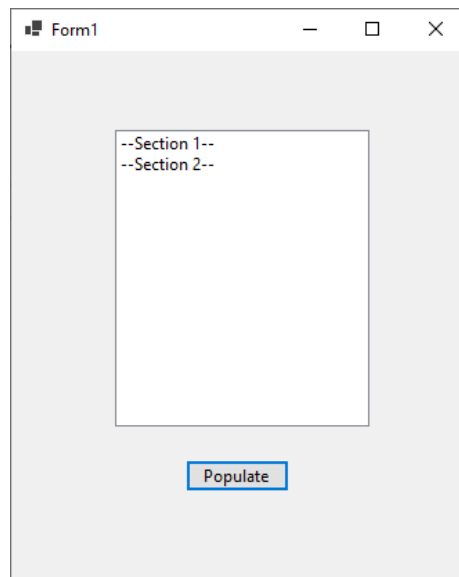


Figure 22. Displaying the sections in the listbox.

Now, we can implement the inner loop. Let's use `foreach` loop this time. Inside the outer loop, we can access to each section element stored in the list through `sections[index]`.

In the first iteration of the outer loop, where index will be 0, `sections[0]` will return the values of { "Fatma", "Hakan", "Furkan" }. In the second iteration, where index will be 1, `sections[1]` will return the values of { "David", "Gustavo", "Alonso" };

Below, you can see the implementation of the `foreach` loop inside the `for` loop. At each iteration of `foreach`, we add student names to the listbox.

```
for (int index = 0; index < sections.Count; index++)
{
    int sectionNo = index + 1;
    lst_studentsBySections.Items.Add("--Section " +
                                    sectionNo.ToString() + "--");

    foreach (string student in sections[index])
    {
        lst_studentsBySections.Items.Add(student);
    }
}
```

If you test your application again, you should obtain the following screen shown in Figure 23.

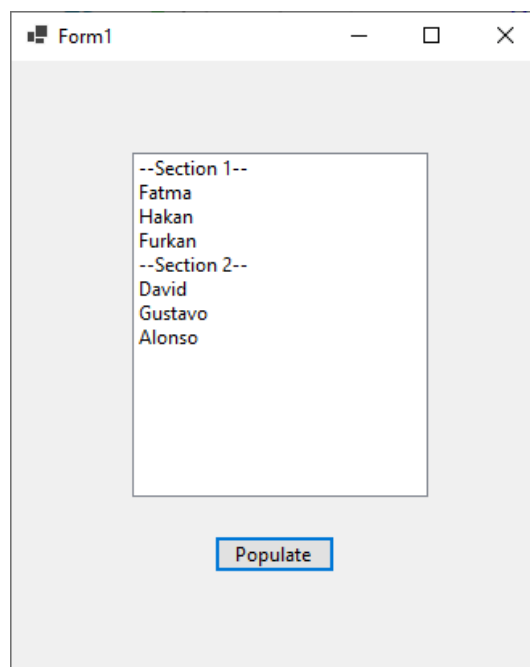


Figure 23. Displaying the sections and students in the listbox.