Module 8 – Strings and Chars.

Often in programming, we need to process text data (or *strings*). A common example is a program that requires a password with certain requirements, such as containing at least one number, special character, and a capital letter. To write such a program, we would need to properly process the possible password provided by the user to make sure it meets these requirements. In this chapter, we will learn about processing textual data. C# offers many tools and techniques for this purpose, which are the focus of this chapter.

To continue with the rest of this chapter, please create a new project called Module8_1.

1) The char data type

`char` is a special data type that is used to define variables to store individual characters, as opposed to the `string` data type which is used to store text. You may remember that to assign a value to a string data type, we use double quotation marks. For `char` type, we enclose the values inside single quotation marks as shown below:

```
char letter = 'a';//for chars, only single quotaion marks are allowed

//since strings CANNOT be assigned to a char type
char letter = "a";//this will throw an error
```

As you can see in the code above, strings cannot be assigned to char type variables (even though if they hold a single character as in the example). `string` and `char` are treated as different data types. We can convert a char variable to a string using the ToString() method. For example, the following statement converts the `letter` variable to a string and assigns to the **Text** property of a label control, named `lbl_message`.

```
lbl_message.Text = letter.ToString();
```

2) Strings as arrays of characters

In C#, a `string` type variable can be treated as an `array` of characters. Below, we define a string called `firstName` initialized with “*Alonso*”. C# allows us to process strings as if they were arrays.

```
string firstName = "Alonso"; //['A', 'l', 'o', 'n', 's', 'o']
```

We can indicate the **index** inside square brackets `[ ]` (also called *subscript*) to access a specific characters inside a string. To access the first character, we need to use index 0, to access the second character, we need to use index 1, and so forth. For example, the statement below assigns the first letter of the `firstName` variable to a `char` variable, `firstLetter`.

```
char firstLetter = firstName[0];
```

Since a string can be treated as an array, we can use loops to iterate through the characters of the string. The following code shows an example in which a comma is placed after each letter of the `firstName` variable, and the resulting `string` is added to the Text property of lbl_message.

```
foreach(char letter in firstName)
{
    lbl_message.Text += letter.ToString() + ", ";
}
```

Please not that square brackets [ ] gives only a read-only access, and it **CANNOT** be used to replace an existing character inside a string.

```
firstLetter[0] = 'B'; //will throw an exception
```

3) Testing a value of a character

Sometimes, we need to check if an individual character in a string is a (uppercase/lowercase) letter, or a number, or a punctation mark. C# provides a variety of methods to check the type of a character.

To begin with, **`char.IsDigit`** is used to check if a character is a digit (0-9). It returns `true` if the character is a digit; otherwise, it returns `false`. Below is an example testing if the 3rd character of a string is a digit:

```
string id = "TC12345";

//checking if the 3rd char is a digit
if (char.IsDigit(id[2]))
{
    //Statements to be executed
}
```

Alternatively, we can pass the whole string variable along with the index of the character that we want to test, as shown below:

```
string id = "TC12345";

if (char.IsDigit(id, 2)) //the string variable, and the index
{
    //Statements to be executed
}
```

We can also check if a character is a letter. To do that, we need to use the **`char.IsLetter`** method. The syntax of this method is the same as the `char.IsDigit` method. Below is an example:

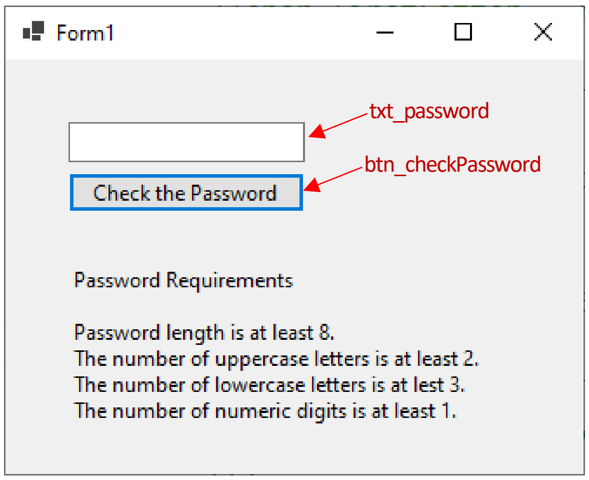
```
string id = "TC12345";

//checking if the 1st char is a letter
if (char.IsLetter(id[0])) //alternative call -> char.IsLetter(id, 0)
{
    //Statements to be executed
}
```

There are some other methods to test for the value of a char type. They are listed in the table below. All of these methods return true or false.

Method	**Explanation**	**Examples**
`char.IsLetterOrDigit`	Returns true only if the character is either letter or a digit.	`char.IsLetterOrDigit('a');` `char.IsLetterOrDigit("ab", 0);`
`char.IsLower`	Returns true if the character is a lowercase letter.	`char.IsLower('a');` `char.IsLower("Ab", 1);`
`char.IsPunctuation`	Returns true if the character is a punctuation mark.	`char.IsPunctuation('a');` `char.IsPunctuation("ab!", 2);`
`char.IsUpper`	Returns true if the character is an uppercase letter.	`char.IsUpper('a');` `char.IsUpper("Ab", 0);`
`char.IsWhiteSpace`	Returns true if the character is a whitespace character (e.g., tab, space).	`char.IsWhiteSpace('a');` `char.IsWhiteSpace("a b", 1);`

We will develop a simple password-requirements checking application to make proper use of these methods. First, we will build the interface shown in Figure 1.



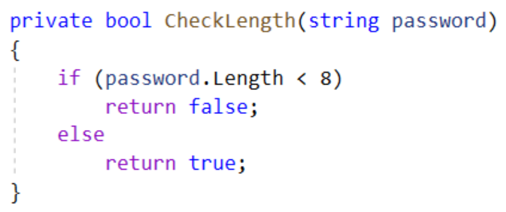
Password Requirements

Password length is at least 8.
The number of uppercase letters is at least 2.
The number of lowercase letters is at lest 3.
The number of numeric digits is at least 1.

Figure 1. Interface of the application.

As you can see in Figure 1, there are four requirements for a valid password entry. We will create a method to check for each of these requirements.

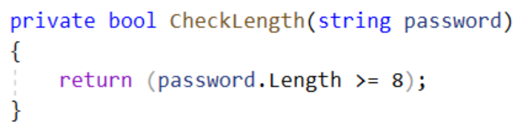
First, let's create a method called **`CheckLength`** to test if the password has the require0d length. This method will return false if the password has less than 8 characters; otherwise, it will return true. The definition of this method is provided in Figure 2.



```
private bool CheckLength(string password)
{
    if (password.Length < 8)
        return false;
    else
        return true;
}
```

Figure 2. Definition of the CheckLength method.

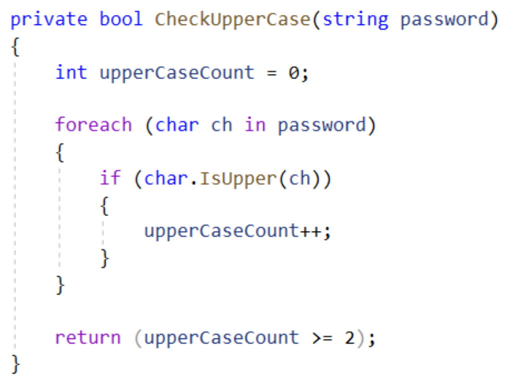
The if condition inside this method can be shortened by directly returning the Boolean expressions as shown in Figure 3.



```
private bool CheckLength(string password)
{
    return (password.Length >= 8);
}
```

Figure 3. An alternative definition of the CheckLength method.

Now, we will define the **`CheckUpperCase`** method, which will return true only if the number of uppercase letters in the password is equal to 2 or more. We will use `foreach` to iterate through the password string.



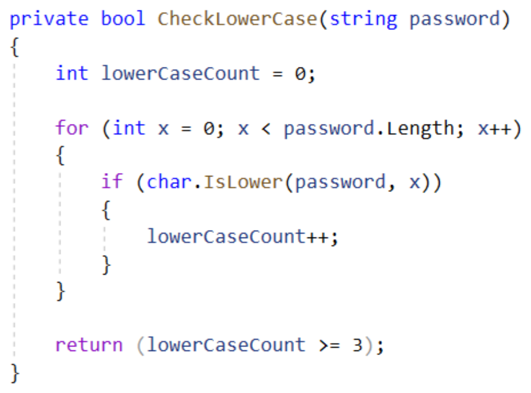
```
private bool CheckUpperCase(string password)
{
    int upperCaseCount = 0;

    foreach (char ch in password)
    {
        if (char.IsUpper(ch))
        {
            upperCaseCount++;
        }
    }

    return (upperCaseCount >= 2);
}
```

Figure 4. Definition of the CheckUpperCase method.

Next, we will define the **`CheckLowerCase`** method, which will return true only if the number of lowercase letters in the password is equal to 3 or more. We will use `for` to iterate through the password string.



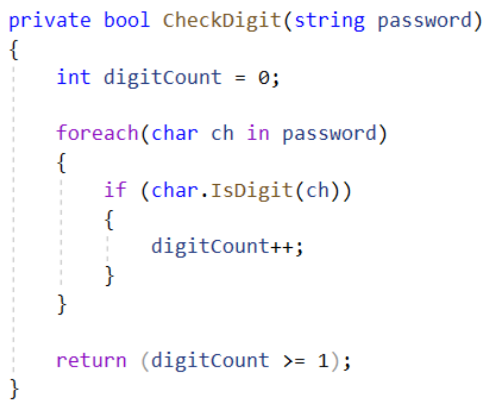
```
private bool CheckLowerCase(string password)
{
    int lowerCaseCount = 0;

    for (int x = 0; x < password.Length; x++)
    {
        if (char.IsLower(password, x))
        {
            lowerCaseCount++;
        }
    }

    return (lowerCaseCount >= 3);
}
```

Figure 5. Definition of the CheckLowerCase method.

Last, we will define the **`CheckDigit`** method, which will return true only if the password contains at least 1 digit. We will use `foreach` to iterate through the password string. See Figure 6 for the definition of the method.



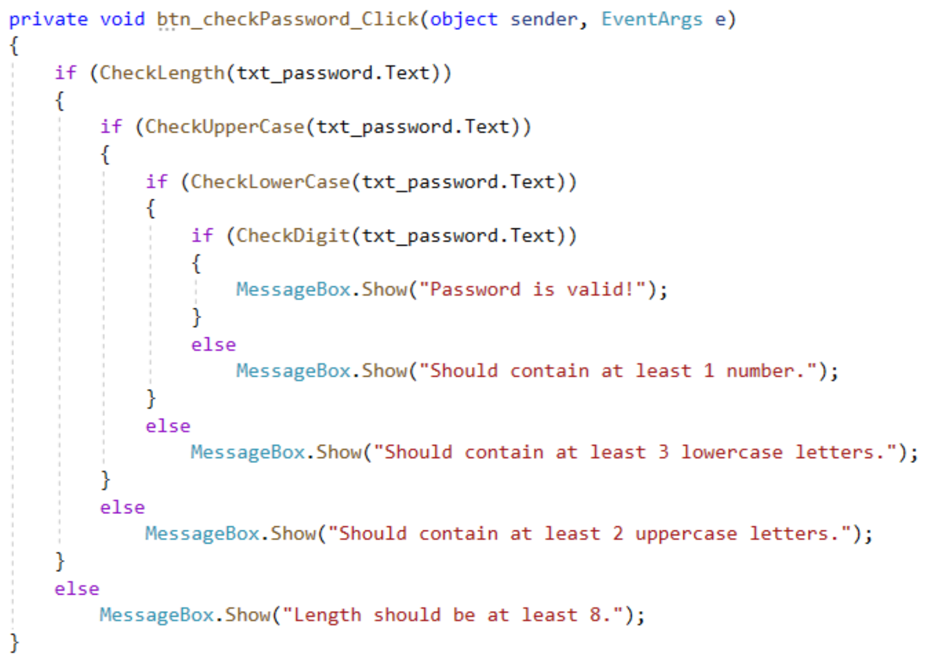
```
private bool CheckDigit(string password)
{
    int digitCount = 0;

    foreach(char ch in password)
    {
        if (char.IsDigit(ch))
        {
            digitCount++;
        }
    }

    return (digitCount >= 1);
}
```

Figure 6. Definition of the CheckDigit method.

Now, please create a click event handler for the button. The methods we created above will be executed when the user clicks on the `btn_checkPassword` button, all method calls should go inside its click event handler. We will use nested if-else statements to check of each requirement (by calling the corresponding method). If any of the requirements is not met, we will print an error message. Only if all the requirements are met, we will print “Password is valid” message. See Figure 7 for the complete code.



```
private void btn_checkPassword_Click(object sender, EventArgs e)
{
    if (CheckLength(txt_password.Text))
    {
        if (CheckUpperCase(txt_password.Text))
        {
            if (CheckLowerCase(txt_password.Text))
            {
                if (CheckDigit(txt_password.Text))
                {
                    MessageBox.Show("Password is valid!");
                }
                else
                    MessageBox.Show("Should contain at least 1 number.");
            }
            else
                MessageBox.Show("Should contain at least 3 lowercase letters.");
        }
        else
            MessageBox.Show("Should contain at least 2 uppercase letters.");
    }
    else
        MessageBox.Show("Length should be at least 8.");
}
```

Figure 7. Click event handler.

4) Working with substrings

Substring is any string within another string. string objects in C# comes with several handy methods to search specific substrings. These methods are:

- `Contains` : This method returns true if a string contains a string or char value.
- `StartsWith` : This method returns true if a string starts with a specific string or char value.
- `EndsWith` : This method returns true if a string ends with a specific string or char value.

The following code shows the use of these methods in action.

```
string title = "Learning C Sharp";

if (title.Contains(“C”)) //returns true
{
    //Statements to be executed
}

if (title.StartsWith(“Learn”)) //returns true
{
    //Statements to be executed
}

if (title.EndsWith(“Harp”)) //returns false
{
    //Statements to be executed
}
```

5) A quick look at Anonymous Methods and Lambda Expressions

The examples above search a string inside another string. Often, we need to search through a list of items and filter them based on their match with a keyword. One example can be searching for users based on their names. Below is a definition of a `list` type variable named **`users`**, with some initial values.

```
List<string> users = new List<string>() { "Onur", "Kerim", "Yannis", "Aynur", "Kerem" };
```

We want to search the users whose names start with K and save the search result in a list object named `searchResult`, as shown below.

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

To find the target users, we will use a loop to traverse through the users list and add them to `searchResult` if they satisfy the search condition. Below is the code to achieve this operation.

```
foreach(string name in users)
{
       if (name.StartsWith("K"))
              searchResult.Append(name);
}
```

Indeed, list objects offers the **Where** method to filter a sequence of values based on a **predicate**. A predicate represents a method that checks whether the parameter provided meets some criteria (e.g., if the string starts with K). A predicate method takes one input parameter and returns a Boolean value: true or false. Below, we define a very simply predicate method, called `StartsWithK`, that returns true if the provided string parameter starts with K:

```
private bool StartsWithK(string name) { return name.StartsWith("K"); }
```

We can pass `StartsWithK` to **Where** method attached to `users` list object, as shown below:

```
users.Where(StartsWithK);
```

Where method will execute the `StartsWithK` method for each single item in the `users` list and keep those items for which `StartsWithK` returns true. We need to convert the resulting object to list by using **ToList** method and assign the whole expression to `searchResult`.The complete code is shown below:

```
searchResult = names.Where(StartsWithK).ToList();
```

Instead of defining the Predicate method separately, we can define an **anonymous** method using the `delegate` keyword. To simplify the process, we can actually place the definition of `StartsWithK` inside the **Where** method directly, as shown below:

```
names.Where(delegate (string name) { return name.StartsWith("K"); })
```

This will function the same way; but it reduces the complexity since it helps achieve the same functionality with less code. **Anonymous** methods do not have a name and they are just created and used for an instant need. Therefore, they are not accessible in any other place in your code.

We could even simplify our code with the help of **Lambda Expressions**. After introduced with C# 3.0, Lambda Expressions supersede Anonymous functions because of their simplicity.

```
searchResult = names.Where(name => name.StartsWith("K")).ToList();
```

In the Lambda Expression, we use a notation (=>) pronounced as “Goes To”. So, on the left hand-side of => we have our input parameter. The lambda expression can infer that the input parameter (whatever its name is) is a string type because Where clause is applied to a list of string objects. The right hand-side of => is the function that evaluates the input parameter based on a condition that returns true or false.

6) Modifying strings

To modify the contents of strings, C# provides various methods:

- `Insert` adds a string into another string at the specified index.
- `Remove` removes part of a string starting from indicated index.

- `ToLower` converts a string to all lowercase characters.
- `ToUpper` converts a string to all uppercase characters.
- `Trim` removes all spaces that appear at the beginning and at the end of a string,
- `TrimStart` removes all spaces at the beginning of a string.
- `TrimEnd` removes all spaces at the end of a string.

Example use of these methods are provided below:

```
"erkan".Insert(0, "S"); //output: Serkan
"erkan".Remove(2); //output: er
"erkan".Remove(2, 1); //output: eran
"Erkan".ToLower(); //output: erkan
"  erkan ".Trim(); //output: “erkan”
"  erkan ".TrimStart(); //output: “erkan “
"  erkan ".TrimEnd (); //output: “  erkan“
```

You might notice that to remove a specific substring from a string we need to know the index of the substring since Remove method only accepts an index parameter. We can use **IndexOf** method to get the index of a specific substring or a character. If the searched substring does not exist, this method returns -1. Below is an example. In this example, we want to delete the ☑ character inside a name. To remove it, we first need to get its index by using **IndexOf** method. Next, we want to add the ✗ character to the end if the name does not already contain it.

```
string name = "Erkan☑";
int index = name.IndexOf("☑");

if (index != -1)
{
    name = name.Remove(index);
}

if (!name.Contains("✗"))
    name = name.Insert(name.Length - 1, "✗");
```

7) Splitting strings into substrings

Sometimes a string can contain multiple values that are separated by a delimiter. Birthdates are good example for such strings. For instance, 14-08-1984 contains day, month, and year information separated by – delimiter. To access the month information, we need to first split the string into smaller chunks. For this purpose, we can use the Split method, which accepts the delimiter character as the parameter. Split method extracts each piece of data separated by the delimiter and returns these pieces inside a string array. Below is the code to split the date string:

```
string date = "14-08-1984";
string [] dateParts = date.Split("-");
```

14 will be stored in index 0, **08** will be stored in index 1, and **1984** will be stored in index 2 of `dateParts` array. That is, `dateParts[0]` will return 14, `dateParts[1]` will return 08 and `dateParts[2]` will return 1984.

Another good example where you may need to use **Split** method is when you have some comma separated values (CSV). Below is `gradeStr` that holds a list of grades separated by comma.

```
string gradesStr = "95,78,87,93,100,83,75,79,66,88";
```

We want to find out how many students achieved a grade higher than 80. To do some math operations, we need to first extract the grades from the string. The following code will save each grade item inside the `grades` array.

```
string[] grades = gradesStr.Split(',');
```

Now, we can write a loop to traverse through string grade values and increase the count variable whenever the grade is greater than 80. The details of the implementation are provided below.

```
int count = 0;
foreach(string g in grades)
{
    if(int.TryParse(g, out int grade))
    {
        if (grade > 80)
            count++;
    }
}
```

If we copy the grades array to a list, we can complete this operation in shorter way. Below code copies the array to a list object named `gradeList`.

```
string gradesStr = "95,78,87,93,100,83,75,79,66,88";
string[] grades = gradesStr.Split(',');
List<int> gradeList = new List<int>();

foreach (string g in grades)
{
    if (int.TryParse(g, out int grade))
    {
        gradeList.Add(grade);
    }
}
```

Once we have the gradeList object ready, we can call its **Count** method. Count will return the number of items in the list. Inside Count, we can write the lambda expression to filter the results, as shown below:

```
gradeList.Count(grade => grade > 80);
```