Session 10. Strings and Regular Expressions

Fundamentally, all data that is input by a user and data that is output to the user is textual data (a string). As a result, the ability to work with and manipulate strings, both for processing input and preparing output, becomes a critical skill in developing any software. In this session, we examine some of the basic mechanisms for creating and manipulating strings, and regular expressions, which are often used in processing input data including validating input and also extracting different parts of that data.

Session Objectives

In this session, you will learn:

- How strings are specified and created in C#;
- How the parts of a string (characters and sub-strings) can be identified, extracted, and compared;
- How to construct larger strings using the StringBuilder class;
- The concept of a regular expression and how they are used to perform advanced matches and replacements against textual data; and
- How to write basic regular expressions and apply them in your programs.

Unit Learning Outcomes

- *ULO 1. Apply object-oriented concepts including abstraction, encapsulation, inheritance, and polymorphism.*
  Although we do not directly address this ULO in this session, we continue developing software that exploit object-oriented concepts, further developing our understanding of these concepts.
- *ULO 2. Solve programming problems using object-oriented techniques and the C# programming language.*
  The application of both strings and regular expressions are skills commonly used in the development of modern applications, and we continue to develop skills relevant to developing object-oriented solutions in the C# programming language.
- *ULO 3. Modify an existing object-oriented application to satisfy new functional requirements.*
  We will not be directly addressing this ULO, however further developing our understanding of the C# programming language contributes to the skills required for this outcome.
- *ULO 4. Construct object-oriented designs and express them using standard UML notation.*
  We will not be directly addressing this ULO, however further developing our understanding of, and practice in, the C# programming language, contributes to improving understanding of object-oriented designs.

Required Reading

Additional reading is required from the textbook Deitel, P., and Deitel, H., "Visual C# 2012: How to Program", Fifth Edition, Pearson, 2014, as follows:

Required Reading: Chapter 16

This chapter covers strings in detail and links to online content covering regular expressions.

Required Tasks

The following tasks are mandatory for this week:

- Task 10.1. Ordered List
- Task 10.2. Regular Expressions

10.1. Introduction

In this session, we take a closer look at working with strings and the use of regular expressions in particular. The ability to work with strings is critical given that almost all input and output to a program is in the form of a string and there is a need to process and produce strings efficiently. Regular expressions are popular for deploying security in modern applications, being one of the tools of choice for performing input validation. As such, regular expressions are quickly becoming a critical element of a programmer’s abilities.

10.2. Strings

The ability to work with strings if a fundamental skill for every programmer – regardless of whether a console or GUI interface is used, all input received from the user and output sent to the user is in a string format. There are a number of fundamental operations on strings that should be possible in any programming language, e.g., copying strings, comparing strings, concatenating strings, and so on. Every programming language, combined with its standard programming library, will usually provide some form of these operations.

10.2.1. String Literals

We have already seen how to specify a string literal in Section 1.5, known as a regular string literal, as follows:

```
"This is a string"
```

This syntax, which also checks for and interprets any escape sequences such as the `"\n"` sequence for a new line (see C for a complete list). This syntax is adequate for most scenarios, however consider what happens if you want to specify a file name on a Microsoft Windows system. In such a file name, any directories in the path are separated by a backslash (‘\’) character, the character used to indicate an escape sequence. The escape sequences allow for this, whereby a double backslash (‘\\’) is interpreted as one. Thus, to specify the following path:

```
C:\Users\ASmith\Documents\Resume.docx
```

we use the following string:

```
"C:\\Users\\ASmith\\Documents\\Resume.docx"
```

Clearly, the support for escape sequences quickly becomes quite painful. To alleviate this, C# provides a second syntax for specifying a string literal, known as a verbatim string literal, where the string is prefixed by the ‘`@`’ symbol. Critically, escape sequences are not interpreted in a verbatim string, allowing us to specify the above file name as follows:

```
@"C:\Users\ASmith\Documents\Resume.docx"
```

Verbatim strings are often used when specifying filenames, SQL queries, and code fragments (such as JavaScript when dynamically building a web page). Verbatim strings are also useful when you want to specify a string containing line breaks in the code, which otherwise results in a syntax error when specifying regular string literals, i.e., the following string produces a syntax error:

```
string badString = "First Line
Second Line
Third Line";
```

however the following string is acceptable:

```
string goodString = @"First Line
Second Line
Third Line";
```

10.2.2. Null and Empty Strings

It is important to remember that the `string` type is in fact a reference type. As such a string variable is a reference, which may or may not have the address of an object stored in it. When no object is being referenced by a string variable, the variable stores a value of `null` and is referred to as a ***null string***. This value can be used to initialise a variable or to test for the presence of an object:

```
string someString = null;
...
if (someString == null)
    // no string to be found
else
    // there is a string
```

Importantly the null string is separate to the concept of an empty string, which is a string object that contains no characters/text, e.g.,

```
string emptyString = "";
```

The empty string can also be referred to using the (static) Empty property of the string class, i.e.,

```
string emptyString = string.Empty;
```

10.2.3. Characters of a String

A string is fundamentally a sequence of characters, and there are occasions where it is useful to consider each character individually. There are a number of ways that we can work with strings in this way. The simplest of these is the indexer operator, which allow us to access the characters of a string in the same way as we work with an array. In the same way as accessing the elements in array, it is important not to exceed the number of characters in the string. The number of characters is determined by using the `Length` property of the `string` object. These concepts are shown in Figure 10.1, which illustrates a simple program that displays each character in a string entered by the user. In the figure the `GetSuffix()` method is used to return an appropriate number suffix for first (st), second (nd), and third (rd) numbers, unless the value is also number in the ‘teens’. A `for` loop is then used to display each character on the screen.

```
static string GetSuffix(int value)
{
    string result = "th";

    if (value % 100 < 11 || value % 100 > 19)
    {
        switch (value % 10)
        {
            case 1:     result = "st";      break;
            case 2:     result = "nd";      break;
            case 3:     result = "rd";      break;
        }
    }

    return result;
}

static void Main(string[] args)
{
    Console.Write("Please enter some text: ");
    string input = Console.ReadLine();

    for (int i = 0; i < input.Length; i++)
    {
        Console.WriteLine("The {0}{1} character is '{2}'", i + 1, GetSuffix(i + 1), 
input[i]);
    }
}
```

Figure 10.1: Example of Accessing Individual Characters in a String

It is also possible to examine each character using a `foreach` loop, as shown in the following code that could replace the for loop in Figure 10.1:

```
foreach (char c in input)
    Console.WriteLine("Character entered: '{0}'", c);
```

It is also possible to copy the contents of a string to an array of characters and back into a string, as follows:

```
string first = "a simple string";
char[] charArray = new char[first.Length];
first.CopyTo(0, charArray, 0, first.Length);
string second = new string(charArray);
```

In this short example, a string first is created with the contents `"a simple string"`, for which an array of characters `charArray` is then created with a number of elements matching the number of characters in the string. The characters in the string are then copied to the array using the `string` class' `CopyTo()` method. Finally, a new `string` is constructed from the array of characters by using one of the string's constructors.

The `CopyTo()` syntax used in the above is relatively straightforward:

```
variable.CopyTo(start_character, destination, start_index, count);
```

In this syntax, *start_character* indicates which is the first character in the string (stored in variable) to be copied to the array. The *destination* parameter indicates the array where characters are to be copied to, storing the characters beginning at *start_index* in the array and continuing for *count* characters.

10.2.4. Other Constructors

Like many classes, the `string` class has a number of constructors that allow us to create strings in various ways. We have already seen one constructor in Section 10.2.3:

```
string second = new string(charArray);
```

This constructor takes a single parameter of an array of characters and generates a string containing those characters. It is also possible to specify which the range of elements in the array to use in the string, as follows:

```
string second = new string(charArray, 0, charArray.GetLength(0));
```

This version is equivalent to the first, except the second parameter indicates the first element to use in the string (`0`, the first array element), and the third parameter indicates the number of elements to copy (the total number of elements in the array).

The other constructor that is useful for most programming tasks was previously introduced in Section 4.9, allows us to create a string consisting of a number of one character, e.g.,

```
string asterisks = new string('*', 40);
```

In this example from Section 4.9, a string is created containing 40 asterisks (`'*'`).

10.2.5. String Comparisons

Comparing two strings together is relatively straightforward in C# through the use of equality operators, e.g.,

```
Console.Write("Please enter the first string: ");
string first = Console.ReadLine();
Console.Write("Please enter the second string: ");
```

```
string second = Console.ReadLine();

if (first == second)
    Console.WriteLine("\"{0}\" is equal to \"{1}\"", first, second);
if (first != second)
    Console.WriteLine("\"{0}\" is not equal to \"{1}\"", first, second);
```

This small example prompts the user for two strings and displays on the `Console` whether they are equal or not equal. Equality can also be checked using the `Equals` method, as follows:

```
if (first.Equals(second))
    Console.WriteLine("\"{0}\" is equal to \"{1}\"", first, second);
```

The above examples are quite limited however, because they only show how to test for equality and inequality. It is also possible to compare two strings to determine their lexicographic order by using the `CompareTo()` method[1], as follows:

```
int result = first.CompareTo(second);
```

The `CompareTo()` method can return one of three values:

- `-1`: the contents of first should appear before second;
- `0`: the contents of first and second are equal; or
- `1`: the contents of first should appear after second.

The lexicographic order is similar to alphabetic order, except for the consideration that the upper case and lower case of characters are seen as separate by the computer, e.g., the words ‘string’ and ‘STRING’ are alphabetically equal but are not lexicographically equal. Lexicographically, lower case letters appear before their upper case equivalents. Consider the following small example code:

```
string lower = "string";
string upper = "STRING";
Console.WriteLine("Result of CompareTo is: {0}", lower.CompareTo(upper));
```

The output of this example code shows a result of `-1`, i.e., `"string"` appears before `"STRING"`.

There is sometimes a need to compare strings alphabetically, instead of lexicographically, which can be achieved as follows:

```
string.Compare(first_string, second_string, true);
```

The `Compare()` method has a number of overloads defined, primarily for allowing the comparison of sub-strings, however this syntax is the most useful. The first two parameters, `first_string` and `second_string`, represent the strings to be compared. The third parameter is a boolean value that indicates whether upper/lower case should be ignored (`true` means to ignore case).

[1] The `CompareTo()` method is actually defined by the `IComparable` interface, which is implemented by the `System.String` type. This interface is also implemented by many other classes defined in Microsoft.Net.

It is also possible to compare the first and last characters of a string, as follows:

```
string testString = "a test string";
if (testString.StartsWith("a test"))
    Console.WriteLine("String starts with \"a test\"");
if (testString.StartsWith("A TEST", true, null))
itive
    Console.WriteLine("String starts with \"A TEST\"");
if (testString.EndsWith("string"))
    Console.WriteLine("String ends with \"string\"");
if (testString.EndsWith("STRING", true, null))
itive
    Console.WriteLine("String ends with \"STRING\"");
```

Finally, it is also possible to test for a sub-string occurring in a string, as follows:

```
string testString = "first >HERE< and last >HERE<";
int indexFromStart = testString.IndexOf("HERE");
int indexFromEnd = testString.LastIndexOf("HERE");
```

Characters can also be located in a string using the same `Indexof()` and `LastIndexOf()` methods, passing the character as a parameter instead of a string.

10.2.6. Creating Strings from Strings

In Section 4.7.2 we introduced string concatenation, where several strings are joined together to create a new string by using the `+` operator, e.g.,

```
string result = "a" + "b" + "c" + "d" + "e";
```

The strings appearing in such a statement be either regular string literals, verbatim string literals, or any combination of the two. Recall also however that this approach is very inefficient, because a new `string` object is potentially created for each `+` operator appearing in the expression, caused by the fact that a `string` object is immutable (cannot be modified).

The solution presented in Section 4.7.2 is to make use of the `string.Format()` method, which allows composite formatting (Section 2.9) to be used to construct a `string`. Importantly, the `Format()` method is just one of many methods defined by the `string` type/class[2], and if only `string` concatenation is required, the `Concat()` method is probably even more efficient, e.g.,

```
string result = string.Concat("a", "b", "c", "d", "e");
```

It is also possible to create strings by extracting sub-strings from another string, achieved using the `Substring()` method. There are two overloads of the `Substring()` method, as follows:

```
variable.Substring(start_index);
variable.Substring(start_index, count);
```

[2] The C# `string` type is actually a shortcut for the Microsoft.Net `System.String` class.

In each case, the `Substring()` method returns a new `string` object containing the substring from variable that begins at *start_index*, and continues for *count* characters (in the case of the second overload).

10.2.7. Other String Manipulations

There are also a number of other useful methods provide for strings, as follows:

- Change the case of the string to upper/lower case:

```
result = someString.ToLower();          // convert to lower case
result = someString.ToUpper();          // convert to upper case
```

- Add "padding" to a string by adding spaces on the left/right side up to a specified field width:

```
result = someString.PadLeft(width);     // adds spaces to LHS
result = someString.PadRight(width);    // adds spaces to RHS
```

- Trim the whitespace characters (space, tab, new line) appearing at the start/end of the string:

```
result = someString.Trim();
```

10.2.8. The StringBuilder Class

As seen in the previous sections in this Session, there are many different manipulations of a string that are possible. However, when applying such manipulations it is critical to consider the impact of the immutable property of the string – each time a string is modified in any way, a new string is created.

The `StringBuilder` class is provided by Microsoft.Net and is very efficient at joining many strings together. Recall from Section 10.2.6 that we identified that the `+` operator for string concatenation is very inefficient, and the `Format()` and `Concat()` methods of the `string` class are preferable. Now consider a problem where the number of strings to be concatenated is not known, or all the strings to be concatenated are not immediately available, e.g., the strings are being entered by the user or read from a network connection. In this situation, we still have the three options, but the operations will need to be repeated until all strings are received. In this situation, many new string objects are still being created.

The `StringBuilder` class provides a good solution to concatenating strings, as it does not allocate a new string each time a string is added. The `StringBuilder` class provides the following useful methods:

- `Append(`*object*`)` – appends the string representation of the specified *object* to the string;
- `AppendFormat(...)` – uses the same syntax as `Console.Write()` or `string.Format()` and allows a string to be added using composite formatting[3];

[3] Note that there is no equivalent of `WriteLine`, however a new line can be added either using the `\n` escape character or by appending the `Environment.NewLine` constant.

- `AppendLine(`*object*`)` – same as `Append()` except the new line character is appended after the string;
- `Insert(`*index*`, `*object*`)` – same as `Append()` except the string is inserted at the indicated *index*;
- `Remove(`*start_index*`, `*length*`)` – remove characters stored in the `StringBuilder` beginning at index *start_index* and continuing for *length* characters;
- `Replace(`*old_string*`, `*new_string*`)` – replace all occurrences of *old_string* with *new_string*; and
- `ToString()` – obtain the final string that has been built using the `StringBuilder` object.

Task 10.1. Ordered List

Objective: Working with strings is an critical skill for any programmer. In this task you will be required to work with some of the methods presented above to build an ordered list of terms.

For this task you are required to write an application that prepares a list of terms. For this purpose, your program should continuously prompt the user for a term until they enter the phrase '`END`'. Upon reading a term, the program will compare the new term against all previously stored terms. If the term has been stored previously, then the term is rejected and the user is prompted for the next term. If the term is a sub-string of a stored term, the stored term should be displayed using the following message:

```
Similar term found: stored_term
```

If the new term isn't rejected, the new term should be stored in the list. Also note the following requirements:

- Terms must be stored in a `List<string>` object;
- Terms must be stored in lower-case;
- Terms must be stored in order using the List's Insert method:
  `someList.Insert(`*index*`, `*item*`);`
  Where *index* is the location in the list to insert, and *item* is the element to be inserted.
- The list should be re-displayed after a term has been added;
- You must use a `for` loop, a `while` loop, or a `do`-`while` loop to check the elements stored in the list; and
- Only one loop should be used when processing the list after a term has been entered (including searching for duplicates, identifying sub-strings, and determining the location for the insertion).

Note: You may find it easier to use several loops initially, but you only need one loop.

10.3. Regular Expressions

Regular expressions are one of the most powerful mechanisms found in modern programming for working with strings. A ***regular expression*** is a sequence of characters that specify a particular pattern that should appear in some text. Consider the following regular expression:

```
^\D?\d+$
```

The above expression can be used to match a string that satisfies the following rules: the string may optionally start with a single character that is not a digit and is then followed by one or more digits, i.e., this regular expression matches the strings `"A1"`, `"$12345"`, `"1"`, and `"12345"`, but not `"A"` or `"1A"`.

10.3.1. Writing Regular Expressions

In general, a regular expression begins with a substring to search for, to which one or more special character sequences are added. For example, given the phrase:

```
the other question to ask them is how to get there?
```

If we wanted to match the word “the” in the above phrase we might be tempted to use the regular expression ‘`the`’, however this would be matched four times (appearing in ‘`the`’, ‘`other`’, ‘`them`’, and ‘`there`’). To help us specify our pattern, we can make use of a character class. A character class specifies which set of characters can be matched by the regular expression. The character classes defined by Microsoft.Net are as follows:

- `[`*character_group*`]` – matches any character that appears inside the square brackets;
  Note: also supports ranges, e.g., [a-z] matches any lower-case characters.
- `[^`*character_group*`]` – matches any character that does not appear inside the square brackets after the caret symbol (`^`);
  Note: also supports ranges, as above.
- `.` – (a full stop) matches any character except a new-line character;
- `\w` – matches any character forming part of a word (including numbers);
- `\W` – matches any character not forming part of a word;
- `\s` – matches any character that is whitespace (space, tab, new line);
- `\S` – matches any character that is not whitespace;
- `\d` – matches any character that is a digit (0-9); and
- `\D` – matches any character that is not a digit.

For example, we can modify the above regular expression to ‘`the[^mr]`’, and it will now only match the first occurrence of ‘`the`’. However, this is not a correct solution, because if the phrase was to be modified to include the word ‘`breathe`’, we would again be incorrectly matching the word ‘`the`’ again. Another solution you may consider is use the regular expression ‘`\Wthe\W`’ or ‘`\sthe\s`’, however neither of these expressions will work because there is no character appearing before the first occurrence of ‘`the`’.

We can also use quantifiers in regular expressions to specify how many times a character should appear. The quantifiers defined by Microsoft.Net include:

- `*` – match zero or more occurrences;
- `+` – match one or more occurrences;

- `?` – match zero or one occurrences;
- `{n}` – match exactly *n* occurrences;
- `{n,}` – match at least *n* occurrences; and
- `{n,m}` – match *n*-*m* occurrences.

Each of the above quantifiers uses what is known as a greedy approach to matching expressions, where the regular expression will match as many occurrences as possible. It is also possible to use a lazy approach to matching, which will match as few occurrences as possible. The lazy approach is selected by adding an extra question mark (‘`?`’) after the above quantifiers, i.e., `*?`, `+?`, `??`, `{n}?`, `{n,}?`, and `{n,m}?`

It is also possible to match a particular position in the string, for which Microsoft.Net defines the following:

- `^` – matches the beginning of the line/string[4];
- `$` – matches the end of the line/string; and
- `\b` – matches a word boundary (words are separated by non-alphanumeric characters).

Finally, it is also possible to group different parts of expressions together by using parenthesis (‘`(`’ and ‘`)`’) and to provide alternative matches by using a pipe/vertical bar symbol (‘`|`’). For our above example, we could match all occurrences of the word ‘the’ by using the expression ‘`\bthe\b`’.

The last point to note is that you may wish to match some of the above characters that are interpreted by regular expressions as part of your string, e.g., you may wish to match currency values requiring the `$` symbol, which in regular expressions matches the end of the line/string. These characters are usually added to the matching text by escaping them with a backslash (‘`\`’) character, i.e., to match a dollar symbol use '`\$`'.

To get an idea of the power of regular expressions, examine the following web site which provides a repository for regular expressions with wide/general applications:

http://regexlib.com/

10.3.2. Matching Regular Expressions

The reason we use regular expressions is test whether a particular string matches a regular expression, extract parts of the string, and/or replace parts of the string. For these purposes Microsoft.Net provides a number of classes and their associated methods. To use this functionality, you first need to add an additional `using` statement at the top of your program:

```
using System.Text.RegularExpressions;
```

The first step in applying a regular expression is to create a `Regex` object, as follows:

[4] Note that regular expressions can be used to match patterns that occur over more than one line, however we do not consider such expressions here.

```
Regex  regex_variable = new Regex(expression);
```

In this syntax, *expression* represents the regular expression, and is usually enclosed in a verbatim string literal (see Section 10.2.1), e.g., `@"\bthe\b"`. The use of a verbatim string literal ensures any escape sequences relevant to the regular expression are not interpreted as normal escape sequences.

Regular expressions are then matched by using the `Match()` method of the `Regex` class, which obtains the first match from the string:

```
Match  match_variable = regex_variable.Match(input_string);
```

It is possible to test for a successful match or not by checking the `Success` variable of the `Match` object, and get the next match using the `NextMatch()` method as follows:

```
while (match_variable.Success)
{
      // code to process an individual match in match_variable
      match_variable = match_variable.NextMatch();
}
```

It is also possible to retrieve all matches in a collection using the following structure:

```
MatchCollection collection_variable = regex_variable.Matches(input_string);
foreach (Match match_variable in collection_variable)
      // code to process an individual match in match_variable
```

Finally, it is also possible to replace a pattern that has been matched by using the `Replace()` method of the `Regex` class, as follows:

```
regex_variable.Replace(input_string, replacement_text);
```

Task 10.2. Regular Expressions

Objective: Regular expressions are commonly found in programs requiring advanced processing/parsing of text input by the user. In particular, regular expressions are becoming very popular for validating input data for security purposes in web applications and other applications requiring a high degree of security, due to their ability to precisely match patterns in text.

You are required to write a program that accepts input from the user and then tests the user's input to see if it matches any of the following criteria using regular expressions:

i. The input is a valid Australian telephone number, in the format of 10 digits separated by minus signs, e.g., dd-dddd-dddd (this format works for both fixed and mobile phone numbers);
ii. The input is a valid currency value, in the format of a dollar ('$') symbol, followed by one or more digits, a full stop ('.'), followed by exactly two digits;
iii. The input is a valid street address, consisting of one or more numbers, followed by at least two non-numeric words, i.e., ddd wwww wwww; and
iv. The input is a valid password, and must have at least three of: upper case characters, lower case characters, numbers, and symbols – you may assume any

symbols on the main numeric keys on your keyboard for this purpose, i.e., shift-1 (‘!’) through shift-0 (‘)’), i.e., ! thru)).

For any of the above expressions that match, display a message on the console indicating the type of input the program thinks it's recognised.