SIT232 - OBJECT ORIENTED DEVELOPMENT

Session 10. Strings and Regular Expressions

Outline

- Session 10. Strings and Regular Expressions
  - Objectives
  - String Literals
  - Null and Empty Strings
  - Characters of a String
  - String Constructors
  - String Comparisons
  - Creating Strings from Strings
  - Other String Manipulations
  - Regular Expressions
  - Applying Regular Expressions

SESSION 10. STRINGS AND REGULAR EXPRESSIONS

Objectives

- At the end of this session you should:
  - Understand how strings are specified and created in C#;
  - Understand how to examine the individual characters in a string and check for the presence of sub-strings;
  - Understand the need for the StringBuilder class, what it is used for and how to apply it in your programs; and
  - Understand the importance of regular expressions, be able to construct regular expressions and apply them in your programs.

String Literals

- We have already seen how to specify regular string literals:

```
"This is a string"
"C:\\Users\\ASmith\\Documents\\Resume.docx"
```

- Sometimes we don't want to interpret escape sequences though, so use **verbatim string literals** instead:

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

String Literals

- Verbatim string literals also allow strings to be broken over lines, i.e.,
  - Syntax error using regular string literals:

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

  - Legal declaration using verbatim string literals:

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

Null and Empty Strings

- Like any other **reference type**, a string variable does not need to *point to an actual object*
  - Can be set to **`null`** instead...

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

Null and Empty Strings

- There is also the empty string however
  - There is a string object
  - There are zero characters in the string...

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

Or

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

Characters of a String

```
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]);
    }
}
```

Accessing individual characters in a string

Characters of a String

- It is also possible to *convert a string to an array of characters and vice*-versa, e.g.,

```
string first = "a simple string";

char[] charArray = new char[first.Length];
first.CopyTo(0, charArray, 0, first.Length);

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

String Constructors

- Like many classes, the string class offers a number of constructors:
  - *Convert character array to string* (previous slide):

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

  - Again, but can *specify which characters*:

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

  - Create a *string containing a number of one character*:

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

String Comparisons

- String comparisons are done using **lexicographical comparison**:
  - Are “string” and “STRING” equal?
  - Not in the computer world!

String Comparisons

- Simple equality comparisons:

```
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);
```

- Alternatively...

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

String Comparisons

- It is also possible to compare strings for their order, not just equality:

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

- CompareTo() returns:
  - -1: first should appear before second
  - 0: first and second are equal
  - 1: first should appear after second

- There is also Compare for case insensitive comparison (returns -1/0/1):

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

String Comparisons

- Comparing the first and last characters of a string:

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

- Testing for the presence of sub-strings:

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

Creating Strings from Strings

- String concatenation

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

- Extracting **sub-string**s:

variable`.Substring(`*start_index*`);`

variable`.Substring(`*start_index*`,` *end_index*`);`

Other String Manipulations

- Change the case to upper/lower case:

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

- Add "padding"/spaces up to specified field width:

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

- Trim the whitespace characters at start/end:

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

The StringBuilder Class

- All mechanisms we have examined to date are very **bad at concatenating** a lot of strings
  - ***+ operator***
  - ***Concat() method***
  - ***Format() method***

- What if you don’t know the number of strings?
- What if you don’t have all the strings?

The StringBuilder Class

- The **StringBuilder** class is best for this

```
StringBuilder sb_variable = new StringBuilder();
...
string result = sb_variable.ToString()
```

- Useful methods:
  - **`Append(`***object***`)`** – appends object to the string;
  - **`AppendFormat(...)`** – append using composite formatting
  - **`AppendLine(`***object***`)`** – same as Append() + new line
  - **`Insert(`***index*`,` *object***`)`** – same as Append() except insert at *index*;
  - **`Remove(`***start_index*`,` *length***`)`** – remove characters from *start_index* continuing for *length* characters; and
  - **`Replace(`***old_string*`,` *new_string***`)`** – replace all occurrences of *old_string* with *new_string*.

Regular Expressions

- Very powerful mechanism for **matching and manipulating strings**
- Example: *No match for non-digit character & digits*

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

  - **Matches:**
    - `A1`
    - `$12345`
    - `1`
    - `12345`
  - **Does not match**:
    - `A`
    - `1A`

Regular Expressions

- Character classes:
  - **[character_group]** – matches any character inside the square brackets;
    - Note: also supports ranges, e.g., [a-z]
  - **[^character_group]** – matches any character not inside the square brackets;
    - Note: also supports ranges.
  - **. – (a full stop) matches any character except new-line;**
  - **\w – matches any character in a word (including numbers);**
  - **\W – matches any character not in a word;**
  - **\s – matches a whitespace character (space/tab/new-line);**
  - **\S – matches any non-whitespace character;**
  - **\d – matches any digit (0-9); and**
  - **\D – matches any non-digit character.**

Regular Expressions

- Greedy quantifiers – match max occurrences
  - *** – 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**
- Lazy quantifiers – match min occurrences
  - Add a ?, i.e., *?, +?, ??, {n}?, {n,}?, {n,m}?

Regular Expressions

- Match a position in the string:
  - **^ – matches the beginning of the line/string;**
  - **$ – matches the end of the line/string; and**
  - **\b – matches a word boundary.**

Applying Regular Expressions

- Additional using statement required:

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

- Create regular expression object:

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

Applying Regular Expressions

- Check for a match against a string:

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

- Process the match/es:

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

- Alternatively:

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

Applying Regular Expressions

- Finally, replacing instead of matching:

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

Summary

- Session 10. Strings and Regular Expressions
  - Objectives
  - String Literals
  - Null and Empty Strings
  - Characters of a String
  - String Constructors
  - String Comparisons
  - Creating Strings from Strings
  - Other String Manipulations
  - Regular Expressions
  - Applying Regular Expressions

Testing

- Consider the following statement:

  string myString = @empty;

  This string is known as

  a) An empty string
  b) A normal string
  c) A null string
  d) A syntax error
  e) None of the above

Testing

The statement Console.WriteLine(""Welcome to SIT232"") will produce the output

a) Welcome to SIT131

b) "Welcome to SIT131"

c) ""Welcome to SIT131""

d) Syntax error

e) Logic error

Testing

Concatenating text means that

a) Two text values are joined together to make a single text value

b) The text is converted to lower case

c) The text is converted to upper case

d) A numeric value is converted to textual data

e) The text is assigned to a control which has a Text property

Testing

.Consider the following statement:

string myString = string.Empty;

This string is known as

a) An empty string

b) A normal string

c) A null string

d) A syntax error

e) None of the above

Testing

In regular expressions, the escape sequence \w will match

a) Any character
b) Any character that is part of a word
c) Whitespace
d) Digits
e) None of the above

Testing

Consider the following statement:

string myString = "null";

This string is known as

a) An empty string

b) A normal string

c) A null string

d) A syntax error

e) None of the above

Testing

The statement

```
string myString = new string('*', 10);
```

will result in

a) A string containing "*10"

b) A string containing "**********"

c) A syntax error

d) None of the above

Testing

In regular expressions, the escape sequence \s will match

a) Any character
b) Any character that is part of a word
c) Whitespace
d) Digits
e) None of the above

Testing

Consider the following statement:

string myString = null;

This string is known as

a) An empty string

b) A normal string

c) A null string

d) A syntax error

e) None of the above

Testing

The use of an @ symbol before a string literal, e.g., @"Object-Oriented" is known as a/an

a) Complex string literal

b) Email address literal

c) Time-stamped string literal

d) Verbatim string literal

e) None of the above

Summary

- Training Videos:
  - C#: StringBuilder