Lecture 11b

Strings, Characters and Regular Expressions

OBJECTIVES

In this lecture you will learn:

To create and manipulate immutable character-string objects of class `string`.

To create and manipulate mutable character-string objects of class `StringBuilder`.

To manipulate character objects of struct `Char`.

To use regular-expression classes `Regex` and `Match`.

To iterate through matches to a regular expression.

OBJECTIVES

To use character classes to match any character from a set of characters.

To use quantifiers to match a pattern multiple times.

To search for complex patterns in text using regular expressions.

To validate data using regular expressions and LINQ.

To modify `strings` using regular expressions and class `Regex`.

Outline

Outline

11b.2 Fundamentals of Characters and Strings

A **character constant** is a character that is represented as an integer value, called a *character code*.

Character constants are established according to the **Unicode character set**.

A string is an object of class `string` in the `System` namespace representing a series of characters.

These characters can be uppercase letters, lowercase letters, digits and various **special characters**.

String literals, also called **string constants**, are written as sequences of characters in double quotation marks.

A declaration can assign a string literal to a string reference.

11b.2 Fundamentals of Characters and Strings (Cont.)

Performance Tip 11b.1

If there are multiple occurrences of the same `string` literal object in an application, a single copy of it will be referenced from each location in the program that uses that `string` literal. It is possible to share the object in this manner, because `string` literal objects are implicitly constant. Such sharing conserves memory.

To avoid excessive backslash characters, it is possible to exclude escape sequences and interpret all the characters in a `string` literally, using the `@` character.

This approach also has the advantage of allowing strings to span multiple lines by preserving all newlines, spaces and tabs.

Outline

Class `string` provides eight constructors. Figure 11b.1 demonstrates the use of three of the constructors.

`String Constructor.cs`

(1 of 2)

```
1  // Fig. 18.1: StringConstructor.cs
2  // Demonstrating string class constructors.
3  using System;
4
5  class StringConstructor
6  {
7     public static void Main( string[] args )
8     {
9        // string initialization
10       char[] characterArray =
11          { 'b', 'i', 'r', 't', 'h', ' ', 'd', 'a', 'y' };
12       string originalString = "Welcome to C# programming!";
13       string string1 = originalString;
14       string string2 = new string( characterArray );
15       string string3 = new string( characterArray, 6, 3 );
16       string string4 = new string( 'C', 5 );
```

Assign a `string` literal to `string` reference `originalString`.

Copy a reference to another `string` literal.

The `string` constructor can take a character array as an argument.

The `string` constructor can take a `char` array and two `int` arguments for starting position and length.

The `string` constructor can take as arguments a character and an `int` specifying the number of times to repeat that character in the `string`.

Fig. 18.1 | `string` constructors. (Part 1 of 2.)

Outline

String Constructor.cs

(2 of 2)

```
17
18       Console.WriteLine( "string1 = " + "\"" + string1 + "\"\n" +
19          "string2 = " + "\"" + string2 + "\"\n" +
20          "string3 = " + "\"" + string3 + "\"\n" +
21          "string4 = " + "\"" + string4 + "\"\n" );
22    } // end Main
23 } // end class StringConstructor
```

```
string1 = "Welcome to C# programming!"
string2 = "birth day"
string3 = "day"
string4 = "CCCCC"
```

Fig. 18.1 | `string` constructors. (Part 2 of 2.)

11b.3 `string` Constructors (Cont.)

Assign a `string` literal to `string` reference `originalString`.

Copy a reference to another `string` literal.

The `string` constructor can take a character array as an argument.

Software Engineering Observation 11b.1

In most cases, it is not necessary to make a copy of an existing string. All strings are immutable—their character contents cannot be changed after they are created.

11b.3 `string` Constructors (Cont.)

The `string` constructor can take a `char` array and two `int` arguments for starting position and length.

The `string` constructor can take as arguments a character and an `int` specifying the number of times to repeat that character in the `string`.

Outline

- The application in Fig. 18.2 presents the `string` indexer and the `string` property `Length`.

```
1  // Fig. 18.2: StringMethods.cs
2  // Using the indexer, property Length and method CopyTo
3  // of class string.
4  using System;
5
6  class StringMethods
7  {
8     public static void Main( string[] args )
9     {
10       string string1 = "hello there";
11       char[] characterArray = new char[ 5 ];
12
13       // output string1
14       Console.WriteLine( "string1: \"" + string1 + "\"" );
15
16       // test Length property
17       Console.WriteLine( "Length of string1: " + string1.Length );
18
19       // loop through characters in string1 and display reversed
20       Console.Write( "The string reversed is: " );
```

StringMethods.cs

(1 of 2)

Property `Length` allows you to determine the number of characters in a string.

Fig. 18.2 | `string` indexer, `Length` property and `CopyTo` method. (Part 1 of 2.)

Outline

```
21
22       for ( int i = string1.Length - 1; i >= 0; i-- )
23          Console.Write( string1[ i ] );
24
25       // copy characters from string1 into characterArray
26       string1.CopyTo( 0, characterArray, 0, characterArray.Length );
27       Console.Write( "\nThe character array is: " );
28
29       for ( int i = 0; i < characterArray.Length; i++ )
30          Console.Write( characterArray[ i ] );
31
32       Console.WriteLine( "\n" );
33    } // end Main
34 } // end class StringMethods
```

```
string1: "hello there"
Length of string1: 11
The string reversed is: ereht olleh
The character array is: hello
```

StringMethods.cs

(2 of 2)

The `string` indexer treats a `string` as an array of `char`s and returns each character at a specific position in the `string`.

The `string` method `CopyTo` copies a specified number of characters from a `string` into a `char` array.

Fig. 18.2 | `string` indexer, `Length` property and `CopyTo` method. (Part 2 of 2.)

Outline

`StringCompare.cs`

(1 of 5)

Property `Length` allows you to determine the number of characters in a string.

The `string` indexer treats a `string` as an array of `chars` and returns each character at a specific position in the `string`.

As with arrays, the first element of a `string` is considered to be at position `0`.

Common Programming Error 11b.1

Attempting to access a character that is outside a `string`'s bounds i.e., an index less than `0` or an index greater than or equal to the `string`'s length) results in an `IndexOutOfRangeException`.

- The `string` method `CopyTo` copies a specified number of characters from a `string` into a `char` array.

When comparing two `string`s, C# simply compares the numeric codes of the characters in the `string`s.

The application in Fig. 18.3 demonstrates the use of method `Equals`, method `CompareTo` and the equality operator (==).

`StringCompare.cs`


```
1  // Fig. 18.3: StringCompare.cs
2  // Comparing strings
3  using System;
4
5  class StringCompare
6  {
7     public static void Main( string[] args )
8     {
9        string string1 = "hello";
10       string string2 = "good bye";
11       string string3 = "Happy Birthday";
12       string string4 = "happy birthday";
13
14       // output values of four strings
15       Console.WriteLine( "string1 = \"" + string1 + "\"" +
16          "\nstring2 = \"" + string2 + "\"" +
17          "\nstring3 = \"" + string3 + "\"" +
18          "\nstring4 = \"" + string4 + "\"\n" );
19
```

Fig. 18.3 | `string` test to determine equality. (Part 1 of 4.)

Outline

StringCompare.cs

(3 of 5)

```
20      // test for equality using Equals method
21      if ( string1.Equals( "hello" ) )
22         Console.WriteLine( "string1 equals \"hello\"" );
23      else
24         Console.WriteLine( "string1 does not equal \"hello\"" );
25
26      // test for equality with ==
27      if ( string1 == "hello" )
28         Console.WriteLine( "string1 equals \"hello\"" );
29      else
30         Console.WriteLine( "string1 does not equal \"hello\"" );
31
32      // test for equality comparing case
33      if ( string.Equals( string3, string4 ) ) // static method
34         Console.WriteLine( "string3 equals string4" );
35      else
36         Console.WriteLine( "string3 does not equal string4" );
37
```

The `string` class's `Equals` method uses a **lexicographical comparison**—comparing the integer Unicode values of character in each `string`.

The overloaded `string` equality operator also uses a lexicographical comparison to compare two `string`s.

`static` method `Equals` is used to compare the values of two `string`s, showing that string comparisons are case-sensetive.

Fig. 18.3 | `string` test to determine equality. (Part 2 of 4.)

Outline

StringCompare.cs

(4 of 5)

```
38       // test CompareTo
39       Console.WriteLine( "\nstring1.CompareTo( string2 ) is " +
40          string1.CompareTo( string2 ) + "\n" +
41          "string2.CompareTo( string1 ) is " +
42          string2.CompareTo( string1 ) + "\n" +
43          "string1.CompareTo( string1 ) is " +
44          string1.CompareTo( string1 ) + "\n" +
45          "string3.CompareTo( string4 ) is " +
46          string3.CompareTo( string4 ) + "\n" +
47          "string4.CompareTo( string3 ) is " +
48          string4.CompareTo( string3 ) + "\n\n" );
49    } // end Main
50 } // end class StringCompare
```

```
string1 = "hello"
string2 = "good bye"
string3 = "Happy Birthday"
string4 = "happy birthday"
```

Fig. 18.3 | `string` test to determine equality. (Part 3 of 4.)

Outline

StringCompare.cs

(5 of 5)

```
string1 equals "hello"
string1 equals "hello"
string3 does not equal string4

string1.CompareTo( string2 ) is 1
string2.CompareTo( string1 ) is -1
string1.CompareTo( string1 ) is 0
string3.CompareTo( string4 ) is 1
string4.CompareTo( string3 ) is -1
```

Fig. 18.3 | `string` test to determine equality. (Part 4 of 4.)

11b.5 Comparing `strings` (Cont.)

Method `Equals` tests any two objects for equality (i.e., checks whether the objects contain identical contents).

The `string` class's `Equals` method uses a **lexicographical comparison**—comparing the integer Unicode values of character in each `string`.

The overloaded `string` equality operator also uses a lexicographical comparison to compare two `string`s.

`static` method `Equals` is used to compare the values of two `string`s, showing that string comparisons are case-sensitive.

11b.5 Comparing `strings` (Cont.)

Method `CompareTo` returns:

- `0` if the `string`s are equal
- A negative value if the calling `string` is less than the argument `string`
- A positive value if the calling `string` is greater than the argument.

Outline

- Figure 11b.4 shows how to test whether a `string` instance begins or ends with a given `string`.

```
1  // Fig. 18.4: StringStartEnd.cs
2  // Demonstrating StartsWith and EndsWith methods.
3  using System;
4
5  class StringStartEnd
6  {
7     public static void Main( string[] args )
8     {
9        string[] strings = { "started", "starting", "ended", "ending" };
10
11       // test every string to see if it starts with "st"
12       for ( int i = 0; i < strings.Length; i++ )
13          if ( strings[ i ].StartsWith( "st" ) )
14             Console.WriteLine( "\"" + strings[ i ] + "\"" +
15                " starts with \"st\"" );
16
17       Console.WriteLine();
18
```

StringStartEnd.cs

(1 of 2)

Method `StartsWith` determines whether a `string` instance starts with the `string` text passed to it as an argument.

Fig. 18.4 | `StartsWith` and `EndsWith` methods. (Part 1 of 2.)

Outline

```
19       // test every string to see if it ends with "ed"
20       for ( int i = 0; i < strings.Length; i++ )
21          if ( strings[ i ].EndsWith( "ed" ) )
22             Console.WriteLine( "\"" + strings[ i ] + "\"" +
23                " ends with \"ed\"" );
24
25       Console.WriteLine();
26    } // end Main
27 } // end class StringStartEnd
```

```
"started" starts with "st"
"starting" starts with "st"

"started" ends with "ed"
"ended" ends with "ed"
```

StringStartEnd.cs

(2 of 2)

Method `EndsWith` determines whether a `string` instance ends with the `string` text passed to it as an argument.

Fig. 18.4 | `StartsWith` and `EndsWith` methods. (Part 2 of 2.)

Outline

- The application in Fig. 18.5 demonstrates some versions of several `string` methods which search for a specified character or substring in a `string`.

`StringIndexMethods.cs`

(1 of 5)

```
1  // Fig. 18.5: StringIndexMethods.cs
2  // Using string-searching methods.
3  using System;
4
5  class StringIndexMethods
6  {
7     public static void Main( string[] args )
8     {
9        string letters = "abcdefghijklmabcdefghijklm";
10       char[] searchLetters = { 'c', 'a', '$' };
11
12       // test IndexOf to locate a character in a string
13       Console.WriteLine( "First 'c' is located at index " +
14          letters.IndexOf( 'c' ) );
15       Console.WriteLine( "First 'a' starting at 1 is located at index " +
16          letters.IndexOf( 'a', 1 ) );
17       Console.WriteLine( "First '$' in the 5 positions starting at 3 " +
18          "is located at index " + letters.IndexOf( '$', 3, 5 ) );
19
```

Method `IndexOf` locates the first occurrence of a character or substring in a `string` and returns its index, or `-1` if it is not found.

Fig. 18.5 | Searching for characters and substrings in `strings`. (Part 1 of 5.)

Outline

StringIndexMethods.cs

(2 of 5)

```
20       // test LastIndexOf to find a character in a string
21       Console.WriteLine( "\nLast 'c' is located at index " +
22          letters.LastIndexOf( 'c' ) );
23       Console.WriteLine( "Last 'a' up to position 25 is located at " +
24          "index " + letters.LastIndexOf( 'a', 25 ) );
25       Console.WriteLine( "Last '$' in the 5 positions starting at 15 " +
26          "is located at index " + letters.LastIndexOf( '$', 15, 5 ) );
27
28       // test IndexOf to locate a substring in a string
29       Console.WriteLine( "\nFirst \"def\" is located at index " +
30          letters.IndexOf( "def" ) );
31       Console.WriteLine( "First \"def\" starting at 7 is located at " +
32          "index " + letters.IndexOf( "def", 7 ) );
33       Console.WriteLine( "First \"hello\" in the 15 positions " +
34          "starting at 5 is located at index " +
35          letters.IndexOf( "hello", 5, 15 ) );
36
```

Method `LastIndexOf` behaves like `IndexOf`, but searches from the end of the string.

`IndexOf` and `LastIndexOf` can take a `string` instead of a character as the first argument.

Fig. 18.5 | Searching for characters and substrings in `strings`. (Part 2 of 5.)

Outline

StringIndexMethods.cs

(3 of 5)

```
37       // test LastIndexOf to find a substring in a string
38       Console.WriteLine( "\nLast \"def\" is located at index " +
39          letters.LastIndexOf( "def" ) );
40       Console.WriteLine( "Last \"def\" up to position 25 is located " +
41          "at index " + letters.LastIndexOf( "def", 25 ) );
42       Console.WriteLine( "Last \"hello\" in the 15 positions " +
43          "ending at 20 is located at index " +
44          letters.LastIndexOf( "hello", 20, 15 ) );
45
46       // test IndexOfAny to find first occurrence of character in array
47       Console.WriteLine( "\nFirst 'c', 'a' or '$' is " +
48          "located at index " + letters.IndexOfAny( searchLetters ) );
49       Console.WriteLine("First 'c', 'a' or '$' starting at 7 is " +
50          "located at index " + letters.IndexOfAny( searchLetters, 7 ) );
51       Console.WriteLine( "First 'c', 'a' or '$' in the 5 positions " +
52          "starting at 7 is located at index " +
53          letters.IndexOfAny( searchLetters, 7, 5 ) );
```

`IndexOf` and `LastIndexOf` can take a `string` instead of a character as the first argument.

Methods `IndexOfAny` and `LastIndexOfAny` take an array of characters as the first argument and return the index of the first occurrence of any of the characters in the array.

Fig. 18.5 | Searching for characters and substrings in `strings`. (Part 3 of 5.)

Outline

```
54
55       // test LastIndexOfAny to find last occurrence of character
56       // in array
57       Console.WriteLine( "\nLast 'c', 'a' or '$' is " +
58          "located at index " + letters.LastIndexOfAny( searchLetters ) );
59       Console.WriteLine( "Last 'c', 'a' or '$' up to position 1 is " +
60          "located at index " +
61          letters.LastIndexOfAny( searchLetters, 1 ) );
62       Console.WriteLine( "Last 'c', 'a' or '$' in the 5 positions " +
63          "ending at 25 is located at index " +
64          letters.LastIndexOfAny( searchLetters, 25, 5 ) );
65    } // end Main
66 } // end class StringIndexMethods
```

StringIndexMethods.cs

(4 of 5)

Methods `IndexOfAny` and `LastIndexOfAny` take an array of characters as the first argument and return the index of the first occurrence of any of the characters in the array.

Fig. 18.5 | Searching for characters and substrings in `strings`. (Part 4 of 5.)

```
First 'c' is located at index 2
First 'a' starting at 1 is located at index 13
First '$' in the 5 positions starting at 3 is located at index -1

Last 'c' is located at index 15
Last 'a' up to position 25 is located at index 13
Last '$' in the 5 positions starting at 15 is located at index -1

First "def" is located at index 3
First "def" starting at 7 is located at index 16
First "hello" in the 15 positions starting at 5 is located at index -1

Last "def" is located at index 16
Last "def" up to position 25 is located at index 16
Last "hello" in the 15 positions ending at 20 is located at index -1

First 'c', 'a' or '$' is located at index 0
First 'c', 'a' or '$' starting at 7 is located at index 13
First 'c', 'a' or '$' in the 5 positions starting at 7 is located at index -1

Last 'c', 'a' or '$' is located at index 15
Last 'c', 'a' or '$' up to position 1 is located at index 0
Last 'c', 'a' or '$' in the 5 positions ending at 25 is located at index -1
```

StringIndexMethods.cs

(5 of 5)

Fig. 18.5 | Searching for characters and substrings in `strings`. (Part 5 of 5.)

11b.6 Locating Characters and Substrings in `strings`

Method `IndexOf` locates the first occurrence of a character or substring in a `string` and returns its index, or `-1` if it is not found.

Method `LastIndexOf` behaves like `IndexOf`, but searches from the end of the string.

`IndexOf` and `LastIndexOf` can take a `string` instead of a character as the first argument.

Methods `IndexOfAny` and `LastIndexOfAny` take an array of characters as the first argument and return the index of the first occurrence of any of the characters in the array.

11b.6 Locating Characters and Substrings in `strings` (Cont.)

Common Programming Error 11b.2

In the overloaded methods `LastIndexOf` and `LastIndexOfAny` that take three parameters, the second argument must be greater than or equal to the third. This might seem counterintuitive, but remember that the search moves from the end of the string toward the start of the string.

Outline

- Class `string` provides two `Substring` methods which create a new `string` by copying part of an existing `string`.
- The application in Fig. 18.6 demonstrates the use of both methods.

`SubString.cs`

(1 of 2)

```
1  // Fig. 18.6: SubString.cs
2  // Demonstrating the string Substring method.
3  using System;
4
5  class SubString
6  {
7     public static void Main( string[] args )
8     {
9        string letters = "abcdefghijklmabcdefghijklm";
10
11       // invoke Substring method and pass it one parameter
```

Fig. 18.6 | Substrings generated from `strings`. (Part 1 of 2.)

Outline

SubString.cs

(2 of 2)

```
12       Console.WriteLine( "Substring from index 20 to end is \"" +
13          letters.Substring( 20 ) + "\"" );
14
15       // invoke Substring method and pass it two parameters
16       Console.WriteLine( "Substring from index 0 of length 6 is \"" +
17          letters.Substring( 0, 6 ) + "\"" );
18    } // end method Main
19 } // end class SubString
```

```
Substring from index 20 to end is "hijklm"
Substring from index 0 of length 6 is "abcdef"
```

The substring returned contains a copy of the characters from the specified starting index to the end of the `string`.

The first argument specifies the starting index, and the second argument specifies the length of the substring to copy.

Fig. 18.6 | Substrings generated from `strings`. (Part 2 of 2.)

- If the starting index is outside the string, or the supplied length of the substring is too large, an `ArgumentOutOfRangeException` is thrown.

Outline

Like the + operator, the `static` method `Concat` of class `string` (Fig. 18.7) concatenates two `string`s and returns a new `string`.

`SubConcatenation.cs`

(1 of 2)

```
1  // Fig. 18.7: SubConcatenation.cs
2  // Demonstrating string class Concat method.
3  using System;
4
5  class StringConcatenation
6  {
7     public static void Main( string[] args )
8     {
9        string string1 = "Happy ";
10       string string2 = "Birthday";
11
12       Console.WriteLine( "string1 = \"" + string1 + "\"\n" +
```

Fig. 18.7 | `Concat static` method. (Part 1 of 2.)

SubConcatenation.cs

(2 of 2)

```
13          "string2 = \"" + string2 + "\"" );
14       Console.WriteLine(
15          "\nResult of string.Concat( string1, string2 ) = " +
16          string.Concat( string1, string2 ) );
17       Console.WriteLine( "string1 after concatenation = " + string1 );
18    } // end Main
19 } // end class StringConcatenation
```

```
string1 = "Happy "
string2 = "Birthday"

Result of string.Concat( string1, string2 ) = Happy Birthday
string1 after concatenation = Happy
```

Append the characters from `string2` to the end of a copy of `string1`, using method `Concat`.

Fig. 18.7 | `Concat static` method. (Part 2 of 2.)

Outline

The application in Fig. 18.8 demonstrates the use of several more `string` methods that return modified copies of a `string`.

`StringMethods2.cs`

(1 of 4)

```
1  // Fig. 18.8: StringMethods2.cs
2  // Demonstrating string methods Replace, ToLower, ToUpper, Trim,
3  // and ToString.
4  using System;
5
6  class StringMethods2
7  {
8     public static void Main( string[] args )
9     {
10       string string1 = "cheers!";
11       string string2 = "GOOD BYE ";
12       string string3 = "   spaces   ";
13
14       Console.WriteLine( "string1 = \"" + string1 + "\"\n" +
15          "string2 = \"" + string2 + "\"\n" +
16          "string3 = \"" + string3 + "\"" );
17
```

Fig. 18.8 | `string` methods `Replace`, `ToLower`, `ToUpper` and `Trim`. (Part 1 of 3.)

Outline

StringMethods2.cs

(2 of 4)

```
18       // call method Replace
19       Console.WriteLine(
20          "\nReplacing \"e\" with \"E\" in string1: \"" +
21          string1.Replace( 'e', 'E' ) + "\"" );
22
23       // call ToLower and ToUpper
24       Console.WriteLine( "\nstring1.ToUpper() = \"" +
25          string1.ToUpper() + "\"\nstring2.ToLower() = \"" +
26          string2.ToLower() + "\"" );
27
28       // call Trim method
29       Console.WriteLine( "\nstring3 after trim = \"" +
30          string3.Trim() + "\"" );
31
32       Console.WriteLine( "\nstring1 = \"" + string1 + "\"" );
33    } // end Main
34 } // end class StringMethods2
```

Method `Replace` returns a new `string`, replacing every occurrence of its first argument with its second argument.

`string` method `ToUpper` generates a new `string` that replaces any lowercase letters with their uppercase equivalents.

Use `string` method `Trim` to remove all whitespace characters that appear at the beginning and end of a `string`.

Method `ToLower` converts a string to lowercase.

Fig. 18.8 | `string` methods `Replace`, `ToLower`, `ToUpper` and `Trim`. (Part 2 of 3.)

Outline

```
string1 = "cheers!"
string2 = "GOOD BYE "
string3 = "   spaces   "

Replacing "e" with "E" in string1: "chEErs!"

string1.ToUpper() = "CHEERS!"
string2.ToLower() = "good bye "

string3 after trim = "spaces"

string1 = "cheers!"
```

StringMethods2.cs

(3 of 4)

Fig. 18.8 | `string` methods `Replace`, `ToLower`, `ToUpper` and `Trim`. (Part 3 of 3.)

Outline

- ✓ Method `Replace` returns a new `string`, replacing every occurrence of its first argument with its second argument.
- ✓ `string` method `ToUpper` generates a new `string` that replaces any lowercase letters with their uppercase equivalents.
- ✓ Method `ToLower` converts a string to lowercase.
- ✓ Use `string` method `Trim` to remove all whitespace characters that appear at the beginning and end of a `string`.
- ✓ `Trim` can also take a character array and return a copy of the `string` that does not begin or end with the characters in the array argument.

`StringMethods2.cs`

(4 of 4)

11b.10 Class `StringBuilder`

Objects of class `string` are immutable.

Class `StringBuilder` is used to create and manipulate dynamic string information—i.e., mutable strings.

`StringBuilder` is much more efficient for working with large numbers of strings than creating individual immutable `string`s

Performance Tip 11b.2

Objects of class string are immutable (i.e., constant strings), whereas objects of class `StringBuilder` are mutable. C# can perform certain optimizations involving `strings` (such as the sharing of one `string` among multiple references), because it knows these objects will not change.

Class `StringBuilderConstructor` (Fig. 18.9) demonstrates three of `StringBuilder`'s six overloaded constructors.

Outline

`StringBuilderConstructor.cs`

(1 of 2)

```
1  // Fig. 18.9: StringBuilderConstructor.cs
2  // Demonstrating StringBuilder class constructors.
3  using System;
4  using System.Text;
5
6  class StringBuilderConstructor
7  {
8     public static void Main( string[] args )
9     {
10       StringBuilder buffer1 = new StringBuilder();
11       StringBuilder buffer2 = new StringBuilder( 10 );
12       StringBuilder buffer3 = new StringBuilder( "hello" );
13
14       Console.WriteLine( "buffer1 = \"" + buffer1 + "\"" );
15       Console.WriteLine( "buffer2 = \"" + buffer2 + "\"" );
16       Console.WriteLine( "buffer3 = \"" + buffer3 + "\"" );
17    } // end Main
18 } // end class StringBuilderConstructor
```

```
buffer1 = ""
buffer2 = ""
buffer3 = "hello"
```

The no-parameter `StringBuilder` constructor creates an empty `StringBuilder` with a default capacity of 16 characters.

Given a single `int` argument, the `StringBuilder` constructor creates an empty `StringBuilder` that has the initial capacity specified in the `int` argument.

Given a single string argument, the `StringBuilder` constructor creates a `StringBuilder` containing the characters of the `string` argument.

Fig. 18.9 | `StringBuilder` class constructors.

Outline

- ✓ The no-parameter `StringBuilder` constructor creates an empty `StringBuilder` with a default capacity of 16 characters.
- ✓ Given a single `int` argument, the `StringBuilder` constructor creates an empty `StringBuilder` that has the initial capacity specified in the `int` argument.
- ✓ Given a single string argument, the `StringBuilder` constructor creates a `StringBuilder` containing the characters of the `string` argument.
  - Its initial capacity is the smallest power of two greater than or equal to the number of characters in the argument `string`, with a minimum of 16.

`StringBuilder Constructor.cs`

(2 of 2)

- ✓ Class `StringBuilder` provides the `Length` and `Capacity` properties.
- ✓ Method `EnsureCapacity` doubles the `StringBuilder` instance's current capacity.
  - ❑ If this doubled value is greater than the value that the programmer wishes to ensure, that value becomes the new capacity.
  - ❑ Otherwise, `EnsureCapacity` alters the capacity to make it equal to the requested number.
- ✓ The program in Fig. 18.10 demonstrates the use of these methods and properties.

Outline

`StringBuilderFeatures.cs`

(1 of 3)

```
1  // Fig. 18.10: StringBuilderFeatures.cs
2  // Demonstrating some features of class StringBuilder.
3  using System;
4  using System.Text;
5
6  class StringBuilderFeatures
7  {
8     public static void Main( string[] args )
```

Fig. 18.10 | `StringBuilder` size manipulation. (Part 1 of 3.)

StringBuilderFeatures.cs

(2 of 3)

```
9     {
10       StringBuilder buffer =
11          new StringBuilder( "Hello, how are you?" );
12
13       // use Length and Capacity properties
14       Console.WriteLine( "buffer = " + buffer +
15          "\nLength = " + buffer.Length +
16          "\nCapacity = " + buffer.Capacity );
17
18       buffer.EnsureCapacity( 75 ); // ensure a capacity of at least 75
19       Console.WriteLine( "\nNew capacity = " +
20          buffer.Capacity );
21
```

Expand the capacity of the `StringBuilder` to a minimum of 75 characters.

Fig. 18.10 | `StringBuilder` size manipulation. (Part 2 of 3.)

```
22       // truncate StringBuilder by setting Length property
23       buffer.Length = 10;
24       Console.Write( "\nNew length = " +
25          buffer.Length + "\nbuffer = " );
26
27       // use StringBuilder indexer
28       for ( int i = 0; i < buffer.Length; i++ )
29          Console.Write( buffer[ i ] );
30
31       Console.WriteLine( "\n" );
32    } // end Main
33 } // end class StringBuilderFeatures
```

```
buffer = Hello, how are you?
Length = 19
Capacity = 32

New capacity = 75

New length = 10
buffer = Hello, how
```

StringBuilderFeatures.cs

(3 of 3)

Use property `Length` to set the length of the `StringBuilder` to 10.

Fig. 18.10 | `StringBuilder` size manipulation. (Part 3 of 3.)

11b.11 Length and Capacity Properties, EnsureCapacity Method and Indexer of Class StringBuilder (Cont.)

When a StringBuilder exceeds its capacity, it grows in the same manner as if method EnsureCapacity had been called.

If Length is set to a value less than the number of characters in the StringBuilder, the contents of the StringBuilder are truncated.

Common Programming Error 11b.3

Assigning `null` to a `string` reference can lead to logic errors if you attempt to compare `null` to an empty `string`. The keyword `null` represents a null reference (i.e., a reference that does not refer to an object), not an empty `string` (which is a `string` object that is of length 0 and contains no characters). The `string.Empty` should be used if you need a `string` with no characters.

Outline

- Class `StringBuilder` provides 19 overloaded `Append` methods that allow various types of values to be added to the end of a `StringBuilder`.
  - The Framework Class Library provides versions for each of the simple types and for character arrays, `string`s and `object`s.
- Figure 11b.11 demonstrates the use of several `Append` methods.

`StringBuilderAppend.cs`

(1 of 3)

```
1  // Fig. 18.11: StringBuilderAppend.cs
2  // Demonstrating StringBuilder Append methods.
3  using System;
4  using System.Text;
5
6  class StringBuilderAppend
7  {
8     public static void Main( string[] args )
9     {
10       object objectValue = "hello";
```

Fig. 18.11 | Append methods of `StringBuilder`. (Part 1 of 3.)

Outline

```
11      string stringValue = "good bye";
12      char[] characterArray = { 'a', 'b', 'c', 'd', 'e', 'f' };
13      bool booleanValue = true;
14      char characterValue = 'Z';
15      int integerValue = 7;
16      long longValue = 1000000;
17      float floatValue = 2.5F; // F suffix indicates that 2.5 is a float
18      double doubleValue = 33.333;
19      StringBuilder buffer = new StringBuilder();
20
21      // use method Append to append values to buffer
22      buffer.Append( objectValue );
23      buffer.Append( " " );
24      buffer.Append( stringValue );
25      buffer.Append( " " );
26      buffer.Append( characterArray );
27      buffer.Append( " ");
28      buffer.Append( characterArray, 0, 3 );
29      buffer.Append( " " );
30      buffer.Append( booleanValue );
```

StringBuilder Append.cs

(2 of 3)

Use 10 different overloaded `Append` methods to attach the string representations of various types to the end of the `StringBuilder`.

Fig. 18.11 | Append methods of `StringBuilder`. (Part 2 of 3.)

Outline

```
31        buffer.Append( " " );
32        buffer.Append( characterValue );
33        buffer.Append( " " );
34        buffer.Append( integerValue );
35        buffer.Append( " " );
36        buffer.Append( longValue );
37        buffer.Append( " " );
38        buffer.Append( floatValue );
39        buffer.Append( " " );
40        buffer.Append( doubleValue );
41
42        Console.WriteLine( "buffer = " + buffer.ToString() + "\n" );
43     } // end Main
44  } // end class StringBuilderAppend
```

```
buffer = hello  good bye  abcdef  abc  True  Z  7  1000000  2.5  33.333
```

StringBuilderAppend.cs

(3 of 3)

Use 10 different overloaded `Append` methods to attach the string representations of various types to the end of the `StringBuilder`.

Fig. 18.11 | Append methods of `StringBuilder`. (Part 3 of 3.)

Outline

Class `StringBuilder`'s method `AppendFormat` converts a `string` to a specified format, then appends it to the `StringBuilder`.

The example in Fig. 18.12 demonstrates the use of `AppendFormat`.

StringBuilder AppendFormat.cs

(1 of 2)

```
1  // Fig. 18.12: StringBuilderAppendFormat.cs
2  // Demonstrating method AppendFormat.
3  using System;
4  using System.Text;
5
6  class StringBuilderAppendFormat
7  {
8     public static void Main( string[] args )
9     {
10       StringBuilder buffer = new StringBuilder();
11
12       // formatted string
13       string string1 = "This {0} costs: {1:C}.\n";
14
15       // string1 argument array
```

The numbers in curly braces specify an index to `objectArray`, passed as the second argument to `appendFormat`.

Fig. 18.12 | `StringBuilder`'s `AppendFormat` method. (Part 1 of 2.)

```
16       object[] objectArray = new object[ 2 ];
17
18       objectArray[ 0 ] = "car";
19       objectArray[ 1 ] = 1234.56;
20
21       // append to buffer formatted string with argument
22       buffer.AppendFormat( string1, objectArray );
23
24       // formatted string
25       string string2 = "Number:{0:d3}.\n" +
26          "Number right aligned with spaces:{0, 4}.\n" +
27          "Number left aligned with spaces:{0, -4}.";
28
29       // append to buffer formatted string with argument
30       buffer.AppendFormat( string2, 5 );
31
32       // display formatted strings
33       Console.WriteLine( buffer.ToString() );
34    } // end Main
35 } // end class StringBuilderAppendFormat
```

```
This car costs: $1,234.56.
Number:005.
Number right aligned with spaces:   5.
Number left aligned with spaces:5   .
```

Outline

StringBuilder
AppendFormat.cs

(2 of 2)

`{0:d3}` specifies that the first argument will be formatted as a three-digit decimal (with leading zeros, if necessary).

`{0, 4}` specifies that the formatted string should have four characters and be right aligned.

`{0: -4}` specifies that the strings should be aligned to the left.

Fig. 18.12 | `StringBuilder`'s `AppendFormat` method. (Part 2 of 2.)

11b.12 Append and AppendFormat Methods of Class StringBuilder (Cont.)

Formats have the form {X[,Y][:FormatString]}.

- X is the number of the argument to be formatted, counting from zero.
- Y is an optional argument, which can be positive or negative, indicating how many characters should be in the result.
- A positive integer aligns the string to the right; a negative integer aligns it to the left.
- The optional FormatString applies a particular format to the argument—currency, decimal or scientific, among others.

One version of AppendFormat takes a string specifying the format and an array of objects to serve as the arguments to the format string.

AppendFormat can take a string containing a format and an object to which the format is applied.

11b.12 Append and AppendFormat Methods of Class StringBuilder (Cont.)

Class `StringBuilder` provides 18 overloaded `Insert` methods to allow various types of data to be inserted at any position in a `StringBuilder`.

Each method inserts its second argument into the `StringBuilder` in front of the character in the position specified by the first argument.

Class `StringBuilder` also provides method `Remove` for deleting any portion of a `StringBuilder`.

Method `Remove` takes two arguments—the index at which to begin deletion and the number of characters to delete.

The `Insert` and `Remove` methods are demonstrated in Fig. 18.13.

```
1  // Fig. 18.13: StringBuilderInsertRemove.cs
2  // Demonstrating methods Insert and Remove of the
3  // StringBuilder class.
4  using System;
5  using System.Text;
6
7  class StringBuilderInsertRemove
8  {
9     public static void Main( string[] args )
10    {
11       object objectValue = "hello";
12       string stringValue = "good bye";
13       char[] characterArray = { 'a', 'b', 'c', 'd', 'e', 'f' };
14       bool booleanValue = true;
15       char characterValue = 'K';
16       int integerValue = 7;
17       long longValue = 10000000;
18       float floatValue = 2.5F; // F suffix indicates that 2.5 is a float
19       double doubleValue = 33.333;
20       StringBuilder buffer = new StringBuilder();
21
```

`StringBuilder InsertRemove.cs`

(1 of 3)

Fig. 18.13 | `StringBuilder` text insertion and removal. (Part 1 of 3.)

Outline

```
22        // insert values into buffer
23        buffer.Insert( 0, objectValue );
24        buffer.Insert( 0, " " );
25        buffer.Insert( 0, stringValue );
26        buffer.Insert( 0, " " );
27        buffer.Insert( 0, characterArray );
28        buffer.Insert( 0, " " );
29        buffer.Insert( 0, booleanValue );
30        buffer.Insert( 0, " " );
31        buffer.Insert( 0, characterValue );
32        buffer.Insert( 0, " " );
33        buffer.Insert( 0, integerValue );
34        buffer.Insert( 0, " " );
35        buffer.Insert( 0, longValue );
36        buffer.Insert( 0, " " );
37        buffer.Insert( 0, floatValue );
38        buffer.Insert( 0, " " );
39        buffer.Insert( 0, doubleValue );
40        buffer.Insert( 0, " " );
41
```

StringBuilder InsertRemove.cs

(2 of 3)

Fig. 18.13 | StringBuilder text insertion and removal. (Part 2 of 3.)

```
42       Console.WriteLine( "buffer after Inserts: \n" + buffer + "\n" );
43
44       buffer.Remove( 10, 1 ); // delete 2 in 2.5
45       buffer.Remove( 4, 4 );  // delete .333 in 33.333
46
47       Console.WriteLine( "buffer after Removes:\n" + buffer );
48    } // end Main
49 } // end class StringBuilderInsertRemove
```

```
buffer after Inserts:
  33.333  2.5  10000000  7  K  True  abcdef  good bye  hello

buffer after Removes:
  33  .5  10000000  7  K  True  abcdef  good bye  hello
```

StringBuilder InsertRemove.cs

(3 of 3)

Fig. 18.13 | `StringBuilder` text insertion and removal. (Part 3 of 3.)

Outline

`Replace` searches for a specified `string` or character and substitutes another `string` or character in its place. Figure 11b.14 demonstrates this method.

`StringBuilder Replace.cs`

(1 of 2)

```
1  // Fig. 18.14: StringBuilderReplace.cs
2  // Demonstrating method Replace.
3  using System;
4  using System.Text;
5
6  class StringBuilderReplace
7  {
8     public static void Main( string[] args )
9     {
10       StringBuilder builder1 =
11          new StringBuilder( "Happy Birthday Jane" );
12       StringBuilder builder2 =
13          new StringBuilder( "good bye greg" );
14
15       Console.WriteLine( "Before replacements:\n" +
16          builder1.ToString() + "\n" + builder2.ToString() );
```

Fig. 18.14 | `StringBuilder` text replacement. (Part 1 of 2.)

Outline

```
17
18       builder1.Replace( "Jane", "Greg" );
19       builder2.Replace( 'g', 'G', 0, 5 );
20
21       Console.WriteLine( "\nAfter replacements:\n" +
22          builder1.ToString() + "\n" + builder2.ToString() );
23    } // end Main
24 } // end class StringBuilderReplace
```

```
Before Replacements:
Happy Birthday Jane
good bye greg

After replacements:
Happy Birthday Greg
Good bye greg
```

StringBuilderReplace.cs

(2 of 2)

This overload of `Replace` replaces all instances of the first character with the second character, beginning at the index specified by the first `int` and continuing for a count specified by the second `int`.

Fig. 18.14 | `StringBuilder` text replacement. (Part 2 of 2.)

- Another overload of this method takes two characters as parameters and
- replaces each occurrence of the first character with the second character.

11b.14 Char Methods

C# provides a concept called a `struct` (short for structure) that is similar to a class.

Unlike classes, `structs` represent value types.

Like classes, `structs` can have methods and properties, and can use the access modifiers `public` and `private`.

`struct` members are accessed via the member access operator (`.`).

The simple types are actually aliases for `struct` types.

All `struct` types derive from class `ValueType`, which in turn derives from `object`.

11b.14 Char Methods (Cont.)

All `struct` types are implicitly `sealed`

- No `virtual` or `abstract` methods
- Members cannot be declared `protected` or `protected internal`.

Outline

- ✓ char is an alias for the `struct` `Char`.
- ✓ Figure 11b.15 demonstrates some `static` methods of the Char struct.

`StaticCharMethods.cs`

(1 of 4)

```
1  // Fig. 18.15: StaticCharMethods.cs
2  // Demonstrates static character-testing methods
3  // from Char struct
4  using System;
5  using System.Windows.Forms;
6
7  namespace StaticCharMethods
8  {
9     public partial class StaticCharMethodsForm : Form
10    {
11       // default constructor
12       public StaticCharMethodsForm()
13       {
14          InitializeComponent();
15       } // end constructor
16
17       // handle analyzeButton_Click
18       private void analyzeButton_Click( object sender, EventArgs e )
```

Fig. 18.15 | Char's `static` character-testing and case-conversion methods. (Part 1 of 4.)

Outline

StaticCharMethods.cs

(2 of 4)

```
19      {
20         // convert string entered to type char
21         char character = Convert.ToChar( inputTextBox.Text );
22         string output;
23
24         output = "is digit: " +
25            Char.IsDigit( character ) + "\r\n";
26         output += "is letter: " +
27            Char.IsLetter( character ) + "\r\n";
28         output += "is letter or digit: " +
29            Char.IsLetterOrDigit( character ) + "\r\n";
30         output += "is lower case: " +
31            Char.IsLower( character ) + "\r\n";
32         output += "is upper case: " +
33            Char.IsUpper( character ) + "\r\n";
34         output += "to upper case: " +
35            Char.ToUpper( character ) + "\r\n";
36         output += "to lower case: " +
```

`Char` method `IsDigit` determines whether a character is defined as a digit.

`IsLetter` determines whether a character is a letter.

`IsLetterOrDigit` determines whether a character is a letter or a digit.

`IsLower` determines whether a character is a lowercase letter.

`IsUpper` determines whether a character is an uppercase letter.

`ToUpper` returns a character's uppercase equivalent, or the original argument if there is no uppercase equivalent.

Fig. 18.15 | Char's `static` character-testing and case-conversion methods. (Part 2 of 4.)

Outline

StaticCharMethods.cs

(3 of 4)

```
37             Char.ToLower( character ) + "\r\n";
38          output += "is punctuation: " +
39             Char.IsPunctuation( character ) + "\r\n";
40          output += "is symbol: " + Char.IsSymbol( character );
41          outputTextBox.Text = output;
42       } // end method analyzeButton_Click
43    } // end class StaticCharMethodsForm
44 } // end namespace StaticCharMethods
```

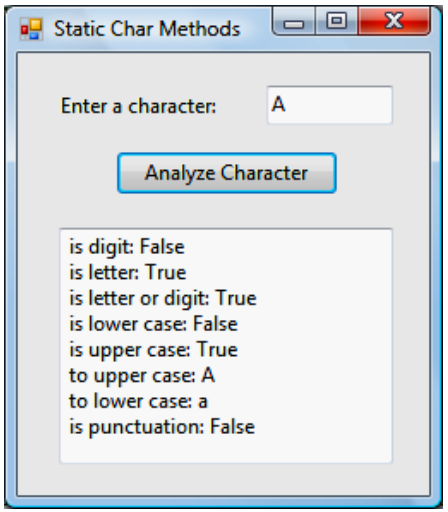
ToLower returns a character lowercase equivalent, or the original argument if there is no lowercase equivalent.

IsPunctuation determines whether a character is a punctuation mark, such as "!", ":" or ")".

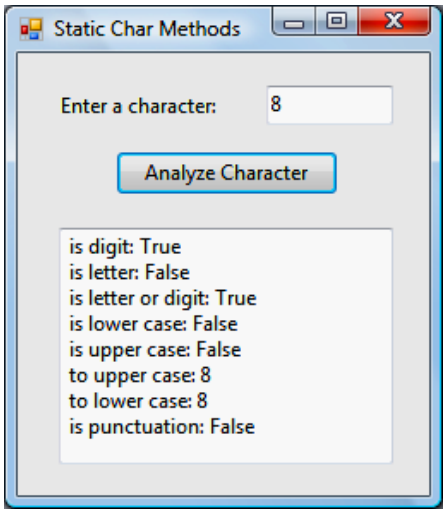
IsSymbol determines whether a character is a symbol, such as "+", "=" or "^".

Fig. 18.15 | Char's `static` character-testing and case-conversion methods. (Part 3 of 4.)

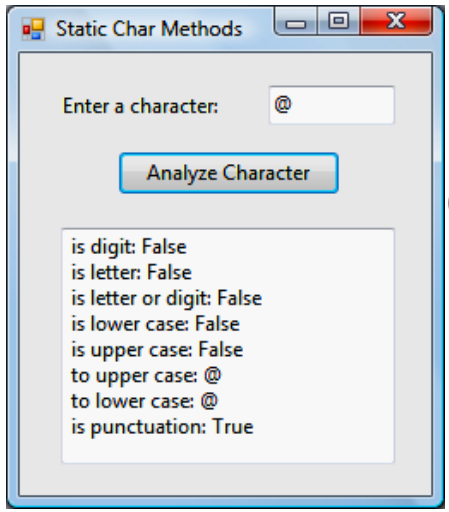
(a)



(b)



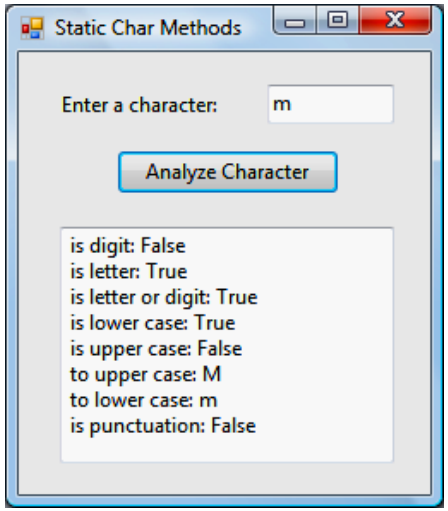
(c)



`StaticCharMethods.cs`

(4 of 4)

(d)



(e)

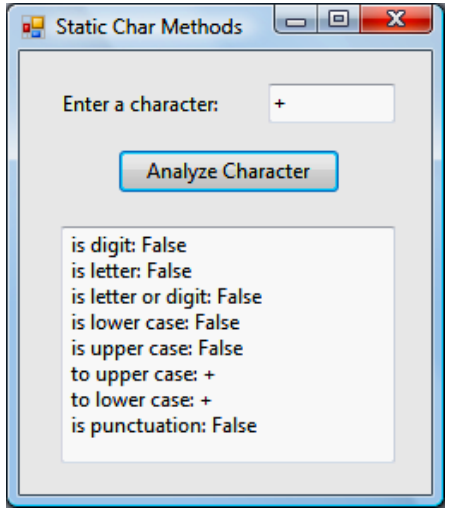


Fig. 18.15 | Char's `static` character-testing and case-conversion methods. (Part 4 of 4.)

11b.14 `Char` Methods (Cont.)

`Char` method `IsDigit` determines whether a character is defined as a digit.

`IsLetter` determines whether a character is a letter.

`IsLetterOrDigit` determines whether a character is a letter or a digit.

`IsLower` determines whether a character is a lowercase letter.

`IsUpper` determines whether a character is an uppercase letter.

`ToUpper` returns a character's uppercase equivalent, or the original argument if there is no uppercase equivalent.

11b.14 Char Methods (Cont.)

`ToLower` returns a character lowercase equivalent, or the original argument if there is no lowercase equivalent.

`IsPunctuation` determines whether a character is a punctuation mark, such as `"!"`, `":"` or `")"`.

`IsSymbol` determines whether a character is a symbol, such as `"+"`, `"="` or `"^"`.

Structure type `Char` contains more `static` methods similar to those shown in this example, such as `IsWhiteSpace`.

The struct also contains several `public` instance methods, many of which we have seen before in other classes, such as `ToString`, `Equals`, and `CompareTo`.

Outline

Class `Card` (Fig. 18.16) contains two `string` instance variables—`face` and `suit`—that store references to the face value and suit name of a specific card.

`Card.cs`

(1 of 2)

```
1  // Fig. 18.16: Card.cs
2  // Stores suit and face information on each card.
3  using System;
4
5  namespace DeckOfCards
6  {
7     public class Card
8     {
9        private string face;
10       private string suit;
11
12       public Card( string faceValue, string suitValue )
13       {
14          face = faceValue;
15          suit = suitValue;
16       } // end constructor
```

The constructor for the class receives two `string`s that it uses to initialize `face` and `suit`.

Fig. 18.16 | `Card` class. (Part 1 of 2.)

Outline

Card.cs

(2 of 2)

```
17
18       public override string ToString()
19       {
20          return face + " of " + suit;
21       } // end method ToString
22    } // end class Card
23 } // end namespace DeckOfCards
```

Method `ToString` (lines 18–21) creates a `string` consisting of the card's `face` and `suit` to identify the card when it is dealt.

Fig. 18.16 | `Card` class. (Part 2 of 2.)

Outline

- We develop the `DeckForm` application (Fig. 18.17), which creates a deck of 52 playing cards, using `Card` objects.

DeckForm.cs

(1 of 6)

```
1  // Fig. 18.17: DeckForm.cs
2  // Simulating card shuffling and dealing.
3  using System;
4  using System.Windows.Forms;
5
6  namespace DeckOfCards
7  {
8     public partial class DeckOfCardsForm : Form
9     {
10       private Card[] deck = new Card[ 52 ]; // deck of 52 cards
11       private int currentCard; // count which card was just dealt
12
13       // default constructor
14       public DeckOfCardsForm()
15       {
16          // Required for Windows Form Designer support
17          InitializeComponent();
18       } // end constructor
19
```

Fig. 18.17 | Card shuffling and dealing simulation. (Part 1 of 6.)

Outline

DeckForm.cs

(2 of 6)

```
20       // handles form at load time
21       private void DeckForm_Load( object sender, EventArgs e )
22       {
23          string[] faces = { "Ace", "Deuce", "Three", "Four", "Five",
24             "Six", "Seven", "Eight", "Nine", "Ten", "Jack", "Queen",
25             "King" };
26          string[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };
27
28          currentCard = -1; // no cards have been dealt
29
30          // initialize deck
31          for ( int i = 0; i < deck.Length; i++ )
32             deck[ i ] = new Card( faces[ i % 13 ], suits[ i / 13 ] );
33       } // end method DeckForm_Load
34
35       // handles dealButton Click
36       private void dealButton_Click( object sender, EventArgs e )
37       {
38          Card dealt = DealCard();
39
```

Each `Card` is instantiated and initialized with two `string`s—one from the `faces` array and one from the `suits` array.

Fig. 18.17 | Card shuffling and dealing simulation. (Part 2 of 6.)

DeckForm.cs

(3 of 6)

```
40          // if dealt card is null, then no cards left
41          // player must shuffle cards
42          if ( dealt != null )
43          {
44             displayLabel.Text = dealt.ToString();
45             statusLabel.Text = "Card #: " + currentCard;
46          } // end if
47          else
48          {
49             displayLabel.Text = "NO MORE CARDS TO DEAL";
50             statusLabel.Text = "Shuffle cards to continue";
51          } // end else
52       } // end method dealButton_Click
53
54       // shuffle cards
55       private void Shuffle()
56       {
57          Random randomNumber = new Random();
58          Card temporaryValue;
59
60          currentCard = -1;
```

Fig. 18.17 | Card shuffling and dealing simulation. (Part 3 of 6.)

Outline

DeckForm.cs

(4 of 6)

```
61
62          // swap each card with randomly selected card (0-51)
63          for ( int i = 0; i < deck.Length; i++ )
64          {
65             int j = randomNumber.Next( 52 );
66
67             // swap cards
68             temporaryValue = deck[ i ];
69             deck[ i ] = deck[ j ];
70             deck[ j ] = temporaryValue;
71          } // end for
72
73          dealButton.Enabled = true; // shuffled deck can now deal cards
74       } // end method Shuffle
75
76       // deal a card if the deck is not empty
77       private Card DealCard()
78       {
79          // if there is a card to deal, then deal it
80          // otherwise signal that cards need to be shuffled by
81          // disabling dealButton and returning null
```

Swap each card with another randomly chosen card.

Fig. 18.17 | Card shuffling and dealing simulation. (Part 4 of 6.)

```
82          if ( currentCard + 1 < deck.Length )
83          {
84             currentCard++; // increment count
85             return deck[ currentCard ]; // return new card
86          } // end if
87          else
88          {
89             dealButton.Enabled = false; // empty deck cannot deal cards
90             return null; // do not return a card
91          } // end else
92       } // end method DealCard
93
94       // handles shuffleButton Click
95       private void shuffleButton_Click(object sender, EventArgs e)
96       {
97          displayLabel.Text = "SHUFFLING...";
98          Shuffle();
99          displayLabel.Text = "DECK IS SHUFFLED";
100      } // end method shuffleButton_Click
101   } // end class DeckForm
102} // end namespace DeckOfCards
```

DeckForm.cs

(5 of 6)

If the `deck` is not empty, return a `Card` object reference; otherwise, it returns `null`.

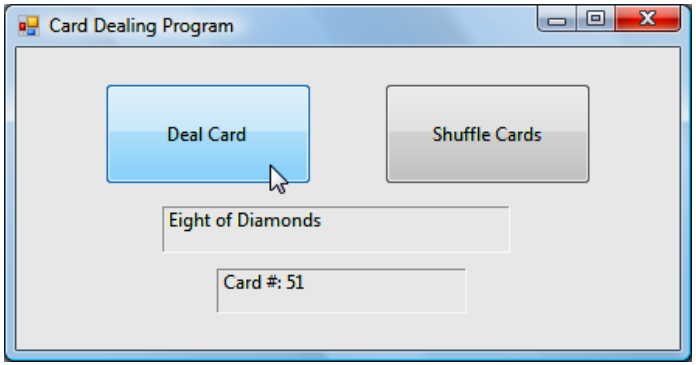
Fig. 18.17 | Card shuffling and dealing simulation. (Part 5 of 6.)

Outline

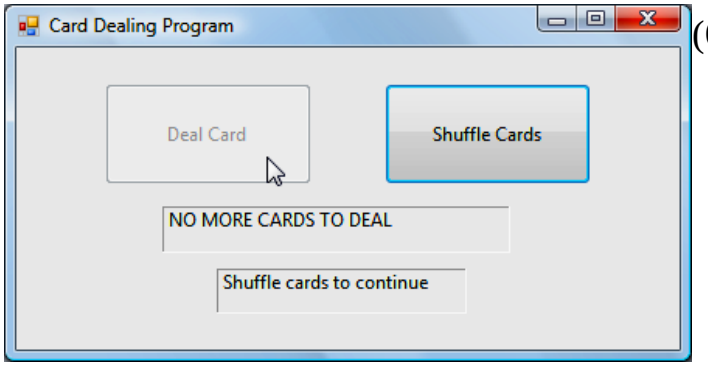
DeckForm.cs

(6 of 6)

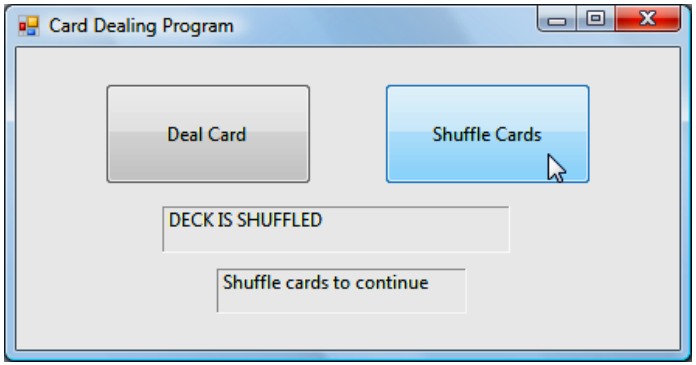
(a)



(b)



(c)



(d)

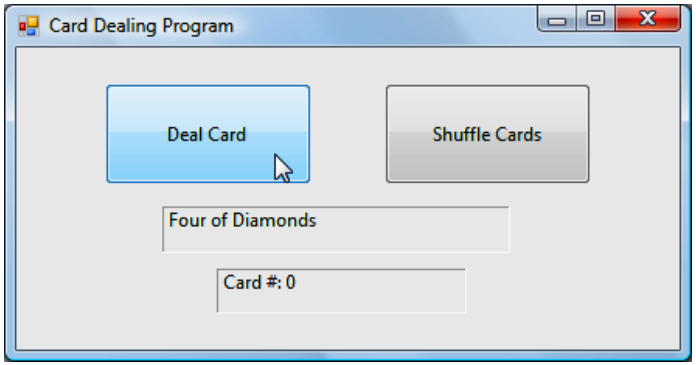


Fig. 18.17 | Card shuffling and dealing simulation. (Part 6 of 6.)

Outline

- **Regular expressions** are specially formatted strings used to find patterns in text.

11b.16.1 Simple Regular Expressions and Class Regex

- Figure 11b.18 demonstrates the basic regular-expression classes.

BasicRegex.cs

(1 of 3)

```
1  // Fig. 18.18: BasicRegex.cs
2  // Demonstrate basic regular expressions.
3  using System;
4  using System.Text.RegularExpressions;
5
6  class BasicRegex
7  {
8     static void Main( string[] args )
9     {
10       string testString =
11          "regular expressions are sometimes called regex or regexp";
12       Console.WriteLine( "The test string is\n   \"{0}\"", testString );
13       Console.Write( "Match 'e' in the test string: " );
14
15       // match 'e' in the test string
16       Regex expression = new Regex( "e" );
17       Console.WriteLine( expression.Match( testString ) );
18       Console.Write( "Match every 'e' in the test string: " );
```

This regular expression matches the literal character `"e"` anywhere in an arbitrary `string`.

Match the leftmost occurrence of the character `"e"` in `testString`.

Fig. 18.18 | Demonstrating basic regular expressions. (Part 1 of 3.)

BasicRegex.cs

(2 of 3)

```
19
20       // match 'e' multiple times in the test string
21       foreach ( var myMatch in expression.Matches( testString ) )
22          Console.Write( "{0} ", myMatch );
23
24       Console.Write( "\nMatch \"regex\" in the test string: " );
25
26       // match 'regex' in the test string
27       foreach ( var myMatch in Regex.Matches( testString, "regex" ) )
28          Console.Write( "{0} ", myMatch );
29
30       Console.Write(
31          "\nMatch \"regex\" or \"regexp\" using an optional 'p': " );
32
33       // use the ? quantifier to include an optional 'p'
34       foreach ( var myMatch in Regex.Matches( testString, "regexp?" ) )
35          Console.Write( "{0} ", myMatch );
36
```

Class `Regex` also provides method `Matches`, which finds all matches and returns a `MatchCollection` object.

The `Regex` static method `Matches` takes a regular expression as an argument in addition to the string to be searched.

Fig. 18.18 | Demonstrating basic regular expressions. (Part 2 of 3.)

Outline

BasicRegex.cs

(3 of 3)

```
37       // use alternation to match either 'cat' or 'hat'
38       expression = new Regex( "(c|h)at" );
39       Console.WriteLine(
40          "\n\"hat cat\" matches {0}, but \"cat hat\" matches {1}",
41          expression.Match( "hat cat" ), expression.Match( "cat hat" ) );
42    } // end Main
43 } // end class BasicRegex
```

Match either "cat" or "hat".

```
The test string is
   "regular expressions are sometimes called regex or regexp"
Match 'e' in the test string: e
Match every 'e' in the test string: e e e e e e e e e e e
Match "regex" in the test string: regex regex
Match "regex" or "regexp" using an optional 'p': regex regexp
"hat cat" matches hat, but "cat hat" matches cat
```

Fig. 18.18 | Demonstrating basic regular expressions. (Part 3 of 3.)

11b.16 Introduction to Regular-Expression Processing

Class `Regex` represents a regular expression.

`Regex` *method* `Match` returns an object of *class* `Match` that represents a single regular-expression match.

Class `Match`'s `ToString` method returns the substring that matched the regular expression.

Class `Regex` also provides method `Matches`, which finds all matches and returns a `MatchCollection` object.

11b.16 Introduction to Regular-Expression Processing (Cont.)

A `MatchCollection` is a collection, similar to an `array`, and can be used with a `foreach` statement to iterate through the collection's elements.

Regular expressions can also be used to match a sequence of literal characters anywhere in a `string`.

The `Regex` `static` method `Matches` takes a regular expression as an argument in addition to the `string` to be searched.

11b.16 Introduction to Regular-Expression Processing (Cont.)

A **metacharacter** is a character with special meaning in a regular expression.

A **quantifier** is a metacharacter that describes how many times a part of the pattern may occur in a match.

The `?` `quantifier` matches zero or one occurrence of the pattern to its left.

The `"|"` (`alternation`) metacharacter matches the expression to its left or to its right.

That the `"|"` character attempts to match the entire expression to its left or to its right.

Alternation chooses the leftmost match in the `string` for either of the alternating expressions.

11b.16 Introduction to Regular-Expression Processing (Cont.)

Regular-Expression Character Classes and Quantifiers

A **character class** represents a group of characters that might appear in a `string`.

The table in Fig. 18.19 lists some character classes that can be used with regular expressions.

Character class	Matches	Character class	Matches
`\d`	any digit	`\D`	any nondigit
`\w`	any word character	`\W`	any nonword character
`\s`	any whitespace	`\S`	any nonwhitespace

Fig. 18.19 | Character classes.

Outline

Figure 11b.20 uses character classes in regular expressions.

CharacterClasses.cs

(1 of 4)

```
1  // Fig. 18.20: CharacterClasses.cs
2  // Demonstrate using character classes and quantifiers.
3  using System;
4  using System.Text.RegularExpressions;
5
6  class CharacterClasses
7  {
8     static void Main( string[] args )
9     {
10       string testString = "abc, DEF, 123";
11       Console.WriteLine( "The test string is: \"{0}\"", testString );
12
13       // find the digits in the test string
14       Console.WriteLine( "Match any digit" );
15       DisplayMatches( testString, @"\d" );
16
17       // find anything that isn't a digit
18       Console.WriteLine( "\nMatch any nondigit" );
19       DisplayMatches( testString, @"\D" );
20
```

We precede the regular expression `string` with `@` to avoid having to escape all backslashes.

Match any character that isn't a digit. Notice in the output that this includes punctuation and whitespace.

Fig. 18.20 | Demonstrating using character classes and quantifiers. (Part 1 of 4.)

Outline

CharacterClasses.cs

(2 of 4)

```
21      // find the word characters in the test string
22      Console.WriteLine( "\nMatch any word character" );
23      DisplayMatches( testString, @"\w" );
24
25      // find sequences of word characters
26      Console.WriteLine(
27         "\nMatch a group of at least one word character" );
28      DisplayMatches( testString, @"\w+" );
29
30      // use a lazy quantifier
31      Console.WriteLine(
32         "\nMatch a group of at least one word character (lazy)" );
33      DisplayMatches( testString, @"\w+?" );
34
35      // match characters from 'a' to 'f'
36      Console.WriteLine( "\nMatch anything from 'a' - 'f'" );
37      DisplayMatches( testString, "[a-f]" );
38
39      // match anything that isn't in the range 'a' to 'f'
40      Console.WriteLine( "\nMatch anything not from 'a' - 'f'" );
41      DisplayMatches( testString, "[^a-f]" );
```

The **+ quantifier** matches one or more occurrences of the pattern to its left.

Create a character class to match any lowercase letter from `a` to `f`.

Matches any character that *isn't* in the range `a-f`.

Fig. 18.20 | Demonstrating using character classes and quantifiers. (Part 2 of 4.)

Outline

CharacterClasses.cs

(3 of 4)

```
42
43       // match any sequence of letters in any case
44       Console.WriteLine( "\nMatch a group of at least one letter" );
45       DisplayMatches( testString, "[a-zA-Z]+" );
46
47       // use the . (dot) metacharacter to match any character
48       Console.WriteLine( "\nMatch a group of any characters" );
49       DisplayMatches( testString, ".*" );
50    } // end Main
51
52    // display the matches to a regular expression
53    private static void DisplayMatches( string input, string expression )
54    {
55       foreach ( var regexMatch in Regex.Matches( input, expression ) )
56          Console.Write( "{0} ", regexMatch );
57
58       Console.WriteLine(); // move to the next line
59    } // end method DisplayMatches
60 } // end class CharacterClasses
```

Use a `foreach` statement to display each `Match` in the `MatchCollection` object returned by `Regex`'s `static` method `Matches`.

Fig. 18.20 | Demonstrating using character classes and quantifiers. (Part 3 of 4.)

Outline

```
The test string is: "abc, DEF, 123"
Match any digit
1 2 3

Match any nondigit
a b c ,   D E F ,

Match any word character
a b c D E F 1 2 3

Match a group of at least one word character
abc DEF 123


Match a group of at least one word character (lazy)
a b c D E F 1 2 3

Match anything from 'a' - 'f'
a b c

Match anything not from 'a' - 'f'
,   D E F ,   1 2 3

Match a group of at least one letter
abc DEF

Match a group of any characters
abc, DEF, 123
```

CharacterClasses.cs

Fig. 18.20 | Demonstrating using character classes and quantifiers. (Part 4 of 4.)

11b.16 Introduction to Regular-Expression Processing (Cont.)

The Negating a character class matches *everything* that *isn't* a member of the character class.

The **+ quantifier** matches one or more occurrences of the pattern to its left.

Quantifiers are **greedy**—they match the *longest* possible occurrence of the pattern.

You can follow a quantifier with a question mark (?) to make it **lazy**—it matches the *shortest* possible occurrence of the pattern.

Figure 11b.21 lists other quantifiers that you can place after a pattern in a regular expression, and the purpose of each.

11b.16 Introduction to Regular-Expression Processing (Cont.)

Quantifier	Matches
`*`	Matches zero or more occurrences of the preceding pattern.
`+`	Matches one or more occurrences of the preceding pattern.
`?`	Matches zero or one occurrences of the preceding pattern.
`.`	Matches any single character.
`{n}`	Matches exactly `n` occurrences of the preceding pattern.
`{n,}`	Matches at least `n` occurrences of the preceding pattern.
`{n,m}`	Matches between n and m (inclusive) occurrences of the preceding pattern.

Fig. 18.21 | Quantifiers used in regular expressions.

11b.16 Introduction to Regular-Expression Processing (Cont.)

You can create your own character class by listing the members of the character class between square brackets, [and].

Metacharacters in square brackets are treated as literal characters.

You can include a range of characters using the "-" character.

You can negate a custom character class by placing a "^" character after the opening square bracket.

11b.16 Introduction to Regular-Expression Processing (Cont.)

You can also use quantifiers with custom character classes.

You can also use the "." (dot) character to match any character other than a newline.

The regular expression ".*" matches any sequence of characters.

The * quantifier matches zero or more occurrences of the pattern to its left.

11b.16.2 Complex Regular Expressions

Outline

The program of Fig. 18.22 tries to match birthdays to a regular expression.

RegexMatches.cs

(1 of 2)

```
1  // Fig. 18.22: RegexMatches.cs
2  // A more complex regular expression.
3  using System;
4  using System.Text.RegularExpressions;
5
6  class RegexMatches
7  {
8     static void Main( string[] args )
9     {
10       // create a regular expression
11       Regex expression = new Regex( @"J.*\d[\d-[4]]-\d\d-\d\d" );
12
13       string testString =
14          "Jane's Birthday is 05-12-75\n" +
15          "Dave's Birthday is 11-04-68\n" +
16          "John's Birthday is 04-28-73\n" +
17          "Joe's Birthday is 12-17-77";
18
```

Use the pattern `[\d-[4]]` to match any digit other than 4.

Fig. 18.22 | A more complex regular expression. (Part 1 of 2.)

Outline

```
19       // display all matches to the regular expression
20       foreach ( var regexMatch in expression.Matches( testString ) )
21          Console.WriteLine( regexMatch );
22    } // end Main
23 } // end class RegexMatches
```

RegexMatches.cs

(2 of 2)

```
Jane's Birthday is 05-12-75
Joe's Birthday is 12-17-77
```

Fig. 18.22 | A more complex regular expression. (Part 2 of 2.)

- ✓ When the "-" character in a character class is followed by a character class, the members of the character class following the "-" are removed from the character class preceding the "-".
- ✓ When using character-class subtraction, the class being subtracted must be the last item in the enclosing square brackets.
- ✓ Instances of the "-" character outside a character class are treated as literal characters.

11b.16.3 Validating User Input with Regular Expressions and LINQ

The application in Fig. 18.23 uses regular expressions to validate name, address and telephone-number information input by a user.

Validate.cs


```
1  // Fig. 18.23: Validate.cs
2  // Validate user information using regular expressions.
3  using System;
4  using System.Linq;
5  using System.Text.RegularExpressions;
6  using System.Windows.Forms;
7
8  namespace Validate
9  {
10    public partial class ValidateForm : Form
11    {
12       public ValidateForm()
13       {
14          InitializeComponent();
15       } // end constructor
16
```

Fig. 18.23 | Validating user information using regular expressions. (Part 1 of 7.)

Validate.cs

```
17      // handles OK Button's Click event
18      private void BtnOK_Click( object sender, RoutedEventArgs e )
19      {
20         // find blank TextBoxes and order by TabIndex
21         var emptyBoxes =
22            from UIElement currentControl in GrdItems.Children
23            where currentControl is TextBox
24            let box = currentControl as TextBox
25            where string.IsNullOrEmpty( box.Text )
26            orderby box.TabIndex
27            select box;
28
29         // if there are any empty TextBoxes
30         if ( emptyBoxes.Count() > 0 )
31         {
32            // display message box indicating missing information
33            MessageBox.Show( "Please fill in all fields",
34               "Missing Information", MessageBoxButton.OK,
35               MessageBoxImage.Error );
36
```

When working with nongeneric collections, such as `Controls`, you must explicitly type the range variable.

If one or more `TextBox`es are empty, the program displays a message to the user that all fields must be filled in before the program can validate the information.

Fig. 18.23 | Validating user information using regular expressions. (Part 2 of 7.)

Outline

Validate.cs

(3 of 7)

```
37             emptyBoxes.First().Focus(); // set focus first empty TextBox
38          } // end if
39          else
40          {
41             // check for invalid input
42             if ( !ValidateInput( TxtLastName.Text,
43                "^[A-Z][a-zA-Z]*$", "Invalid last name" ) )
44
45                TxtLastName.Focus(); // select invalid TextBox
46             else if ( !ValidateInput(TxtFirstName.Text,
47                "^[A-Z][a-zA-Z]*$", "Invalid first name" ) )
48
49                TxtFirstName.Focus(); // select invalid TextBox
50             else if ( !ValidateInput( TxtAddress.Text,
51                @"^[0-9]+\s+([a-zA-Z]+|[a-zA-Z]+\s[a-zA-Z]+)$",
52                "Invalid address" ) )
53
54                TxtAddress.Focus(); // select invalid TextBox
55             else if ( !ValidateInput(TxtCity.Text,
56                @"^([a-zA-Z]+|[a-zA-Z]+\s[a-zA-Z]+)$", "Invalid city" ) )
57
```

The address can contain a word of one or more characters *or* a word of one or more characters followed by a space and another word of one or more characters.

Fig. 18.23 | Validating user information using regular expressions. (Part 3 of 7.)

Outline

Validate (4 of 7)

```
58                TxtCity.Focus(); // select invalid TextBox
59             else if ( !ValidateInput( TxtState.Text,
60                @"^([a-zA-Z]+|[a-zA-Z]+\s[a-zA-Z]+)$", "Invalid state" ) )
61
62                TxtState.Focus (); // select invalid TextBox
63             else if ( !ValidateInput(TxtZipCode.Text,
64                @"^\d{4}$", "Invalid zip code" ) )
65
66                TxtZipCode.Focus(); // select invalid TextBox
67             else if ( !ValidateInput( TxtPhone.Text,
68                @"^(02-\d{4}-\d{3}|0\d{3}-\d{3}-\d{3})$"",
69                "Invalid phone number" ) )
70
71                TxtPhone.Focus(); // select invalid TextBox
72             else // if all input is valid
73             {
74                this.Hide(); // hide main window
75                MessageBox.Show( "Thank You!", "Information Correct",
76                   MessageBoxButton.OK, MessageBoxImage.Information );
77                System.Environment.Exit(0); // exit the application
78             } // end else
79          } // end else
80       } // end method okButton_Click
```

Fig. 18.23 | Validating user information using regular expressions. (Part 4 of 7.)

Outline

Validate.cs

(5 of 7)

```
81
82       // use regular expressions to validate user input
83       private bool ValidateInput(
84          string input, string expression, string message )
85       {
86          // store whether the input is valid
87          bool valid = Regex.Match( input, expression ).Success;
88
89          // if the input doesn't match the regular expression
90          if ( !valid )
91          {
92             // signal the user that input was invalid
93             MessageBox.Show( message, "Invalid Input",
94                MessageBoxButton.OK, MessageBoxImage.Error );
95          } // end if
96
97          return valid; // return whether the input is valid
98       } // end method ValidateInput
99    } // end class ValidateForm
100} // end namespace Validate
```

Call `Regex` `static` method `Match`, passing both the `string` to validate and the regular expression as arguments.

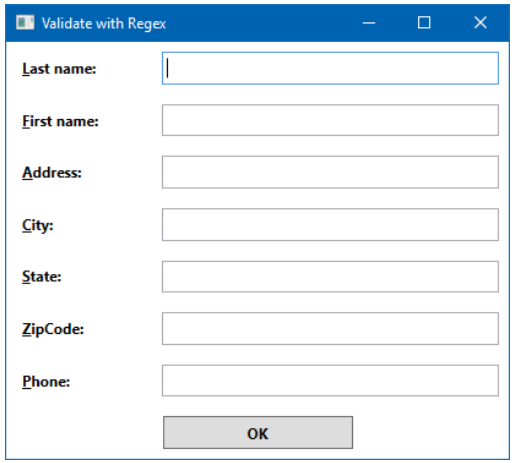
Fig. 18.23 | Validating user information using regular expressions. (Part 5 of 7.)

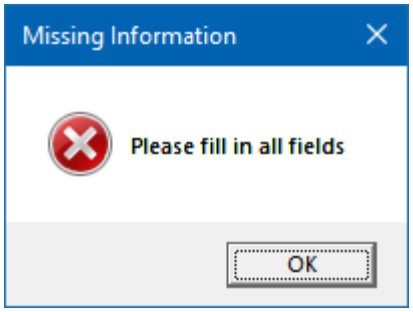
Outline

`Validate.cs`

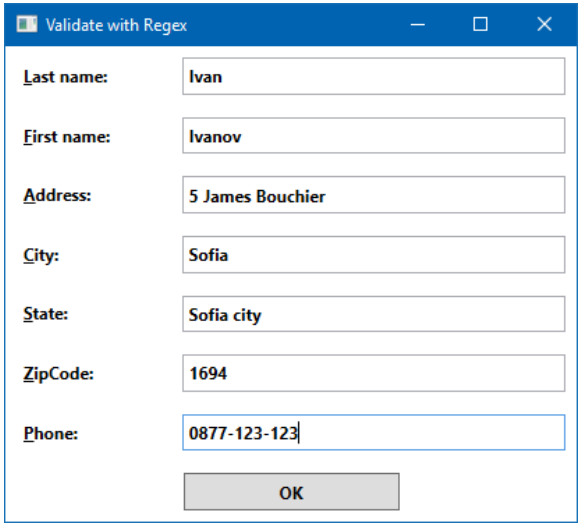
(6 of 7)

(a)





(b)



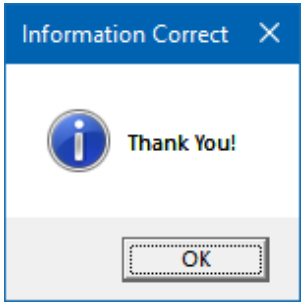


Fig. 18.23 | Validating user information using regular expressions. (Part 6 of 7.)

11b.16 Introduction to Regular-Expression Processing (Cont.)

When working with nongeneric collections, such as `Controls`, you must explicitly type the range variable.

The `let` clause creates and initializes a variable in a LINQ query for use later in the query.

You may include a second `where` clause after the `let` clause.

The `Success` property of class `Match` indicates whether the Match method found a match.

11b.16 Introduction to Regular-Expression Processing (Cont.)

If a regular expression begins with `"^"` and ends with `"$"`, the characters `"^"` and `"$"` represent the beginning and end of a `string`, respectively.

These characters force a regular expression to return a match only if the entire `string` being processed matches the regular expression.

The `\s` character class matches a single whitespace character.

The pattern to the left of `{n}` must occur exactly `n` times.

11b.16 Binding RegEx in XAML

Alternatively, you can create a `class Customer` with properties `LastName`, `FirstName`,…, `Phone` and define a custom `class RegexValidationRule` inheriting from class `ValidationRule` as we used to do when binding properties (study project **WpfRegexValidationRule**)

You may use `Style` and `Validation.ErrorTemplate` attributes of a `TextBox` with to control how the error message is being displayed in case the regular expression fails the validation process.

11b.16 Binding RegEx in XAML

```
public  class Customer
    {
        public string LastName { get; set; } = string.Empty;// "Ivanov";
        public string FirstName { get;  set; } = string.Empty;//"Ivan";
        public string Email { get;  set; } = string.Empty;//"Ivan.Ivanov@first.edu.bg";
        public string City { get;  set; } = string.Empty;//"Ivanovo city";
        public string Address { get;  set; } = string.Empty;// "2 Ivanovo street";
        public string ZipCode { get;  set; } = string.Empty;//"1234";
        public string Phone { get;  set; } = string.Empty;// "02-1234-567";
        public string State { get;  set; } = string.Empty;//"Ivanovo state";
    }
```

```
public class RegexValidationRule : ValidationRule
    {
        public string ErrorMessage { get; set; }
        public RegexOptions RegexOptions { get; set; } = RegexOptions.None;
        public string RegexText { get; set; }
        public override ValidationResult Validate(object value, CultureInfo cultureInfo)
        {
            ValidationResult result = ValidationResult.ValidResult;

            // If there is no regular expression to evaluate,
            // then the data is considered to be valid.
            if (!String.IsNullOrEmpty(this.RegexText))
            {
                // Cast the input value to a string (null becomes empty string).
                string text = value as string ?? String.Empty;

                // If the string does not match the regex, return a value
                // which indicates failure and provide an error mesasge.
                if (!Regex.IsMatch(text, this.RegexText, this.RegexOptions))
                    result = new ValidationResult(false, this.ErrorMessage);
            }
            return result;
        }
    }
```

11b.16 Binding RegEx in XAML

This way we can create a reusable means of validating user input via regular expressions in XAML. The following shows the XAML code needed to validate the Phone property.

```
<TextBox x:Name="TxtPhone"   Grid.Row="6" Grid.Column="1" >
      <TextBox.Text >
          <Binding Path="Phone"  UpdateSourceTrigger="PropertyChanged">
               <Binding.ValidationRules>
                   <local:RegexValidationRule
                      RegexText= "^(02-\d{4}-\d{3}|0\d{3}-\d{3}-\d{3})$"
                      ErrorMessage="Invalid Phone"
                      RegexOptions="IgnoreCase"
                  />
              </Binding.ValidationRules>
          </Binding>
     </TextBox.Text
</TextBox>
```

11b.16.4 Regex Methods Replace and Split

Outline

Class `Regex` provides `static` and instance versions of methods `Replace` and `Split`, which are demonstrated in Fig. 18.24.

RegexSubstitution.cs

(1 of 3)

```
1  // Fig. 18.24: RegexSubstitution.cs
2  // Using Regex methods Replace and Split.
3  using System;
4  using System.Text.RegularExpressions;
5
6  class RegexSubstitution
7  {
8     static void Main( string[] args )
9     {
10       string testString1 = "This sentence ends in 5 stars *****";
11       string testString2 = "1, 2, 3, 4, 5, 6, 7, 8";
12       Regex testRegex1 = new Regex( @"\d" );
13       string output = string.Empty;
14
15       Console.WriteLine( "First test string: {0}", testString1 );
16
```

Fig. 18.24 | Using `Regex` methods `Replace` and `Split`. (Part 1 of 3.)

Outline

RegexSubstitution.cs

(2 of 3)

```
17       // replace every '*' with a '^' and display the result
18       testString1 = Regex.Replace( testString1, @"\*", "^" );
19       Console.WriteLine( "^ substituted for *: {0}", testString1 );
20
21       // replace the word "stars" with "carets" and display the result
22       testString1 = Regex.Replace( testString1, "stars", "carets" );
23       Console.WriteLine( "\"carets\" substituted for \"stars\": {0}",
24          testString1 );
25
26       // replace every word with "word" and display the result
27       Console.WriteLine( "Every word replaced by \"word\": {0}",
28          Regex.Replace( testString1, @"\w+", "word" ) );
29
30       Console.WriteLine( "\nSecond test string: {0}", testString2 );
31
32       // replace the first three digits with the word "digit"
33       Console.WriteLine( "Replace first 3 digits by \"digit\": {0}",
34          testRegex1.Replace( testString2, "digit", 3 ) );
35
36       Console.Write( "string split at commas [" );
37
```

A `static` version of the `Replace` method takes the `string` to modify, the regular expression `string`, and the replacement `string`.

`Replace` is also an instance method that uses the regular expression passed to the constructor of the calling `Regex` object.

Fig. 18.24 | Using `Regex` methods `Replace` and `Split`. (Part 2 of 3.)

Outline

RegexSubstitution.cs

(3 of 3)

```
38       // split the string into individual strings, each containing a digit
39       string[] result = Regex.Split( testString2, @",\s" );
40
41       // add each digit to the output string
42       foreach( var resultString in result )
43          output += "\"" + resultString + "\", ";
44
45       // delete ", " at the end of output string
46       Console.WriteLine( output.Substring( 0, output.Length - 2 ) + "]" );
47    } // end Main
48 } // end class RegexSubstitution
```

The first argument of the `static` `Split` is the `string` to split; the second argument is the regular expression that represents the delimiter.

Fig. 18.24 | Using `Regex` methods `Replace` and `Split`. (Part 3 of 3.)

11b.16 Introduction to Regular-Expression Processing (Cont.)

`Regex` method `Replace` replaces text in a `string` with new text wherever the original `string` matches a regular expression.

A `static` version of the `Replace` method takes the `string` to modify, the regular expression `string`, and the replacement `string`.

`Replace` is also an instance method that uses the regular expression passed to the constructor of the calling `Regex` object.

11b.16 Introduction to Regular-Expression Processing (Cont.)

Method `Split` returns an `array` containing the substrings of a string delimited by matches of a regular expression.

The first argument of the `static` `Split` is the `string` to split; the second argument is the regular expression that represents the delimiter.