16.15 Introduction to Regular-Expression Processing

This section introduces **regular expressions**—specially formatted strings used to find patterns in text. They can be used to ensure that data is in a particular format. For example, a U.S. zip code must consist of five digits, or five digits followed by a dash followed by four more digits. Compilers use regular expressions to validate program syntax. If the program code does not match the regular expression, the compiler indicates that there's a syntax error. We discuss classes `Regex` and `Match` from the `System.Text.RegularExpressions` namespace as well as the symbols used to form regular expressions. We then demonstrate how to find patterns in a string, match entire strings to patterns, replace characters in a string that match a pattern and split strings at delimiters specified as a pattern in a regular expression.

16.15.1 Simple Regular Expressions and Class Regex

The .NET Framework provides several classes to help developers manipulate regular expressions. Figure 16.16 demonstrates the basic regular-expression classes. To use these classes, add a `using` statement for the namespace `System.Text.RegularExpressions` (line 4). Class **`Regex`** represents a regular expression. We create a `Regex` object named `expression` (line 16) to represent the regular expression `"e"`. This regular expression matches the literal character `"e"` anywhere in an arbitrary `string`. `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. The call to method `Match` (line 17) matches the leftmost occurrence of the character `"e"` in `testString`. Class `Regex` also provides method **`Matches`** (line 21), which finds all matches of the regular expression in an arbitrary `string` and returns a `MatchCollection` object containing all the `Match`es. A **`MatchCollection`** is a collection, similar to an `array`, and can be used with a `foreach` statement to iterate through the collection's elements. We introduced collections in Chapter 9 and discuss them in more detail in Chapter 23, Collections. We use a `foreach` statement (lines 21–22) to display all the matches to `expression` in `testString`. The elements in the `MatchCollection` are `Match` objects, so the `foreach` statement infers variable `myMatch` to be of type `Match`. For each `Match`, line 22 outputs the text that matched the regular expression.

```
1  // Fig. 16.16: 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: " );
```

Fig. 16.16 | Demonstrating basic regular expressions. (Part 1 of 2.)

```
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: " );
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
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
```

```
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. 16.16 | Demonstrating basic regular expressions. (Part 2 of 2.)

Regular expressions can also be used to match a sequence of literal characters anywhere in a `string`. Lines 27–28 display all the occurrences of the character sequence `"regex"` in `testString`. Here we use the `Regex` `static` method `Matches`. Class `Regex` provides `static` versions of both methods `Match` and `Matches`. The `static` versions take a regular expression as an argument in addition to the `string` to be searched. This is useful when you want to use a regular expression only once. The call to method `Matches` (line 27) returns two matches to the regular expression `"regex"`. Notice that `"regexp"` in the `testString` matches the regular expression `"regex"`, but the `"p"` is excluded. We use the regular expression `"regexp?"` (line 34) to match occurrences of both `"regex"` and `"regexp"`. The question mark (?) is a **metacharacter**—a character with special meaning in a regular expression. More specifically, the question mark is a **quantifier**—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. In line 34, we apply the ? quantifier to the character "p". This means that a match to the regular expression contains the sequence of characters "regex" and may be followed by a "p". Notice that the foreach statement (lines 34–35) displays both "regex" and "regexp".

Metacharacters allow you to create more complex patterns. The "|" (**alternation**) metacharacter matches the expression to its left or to its right. We use alternation in the regular expression "(c|h)at" (line 38) to match either "cat" or "hat". Parentheses are used to group parts of a regular expression, much as you group parts of a mathematical expression. The "|" causes the pattern to match a sequence of characters starting with either "c" or "h", followed by "at". The "|" character attempts to match the entire expression to its left or to its right. If we didn't use the parentheses around "c|h", the regular expression would match either the single character "c" or the sequence of characters "hat". Line 41 uses the regular expression (line 38) to search the strings "hat cat" and "cat hat". Notice in the output that the first match in "hat cat" is "hat", while the first match in "cat hat" is "cat". Alternation chooses the leftmost match in the string for either of the alternating expressions—the order of the expressions doesn't matter.

Regular-Expression Character Classes and Quantifiers

The table in Fig. 16.17 lists some character classes that can be used with regular expressions. A **character class** represents a group of characters that might appear in a string. For example, a **word character** (**\w**) is any alphanumeric character (a-z, A-Z and 0-9) or underscore. A **whitespace character** (**\s**) is a space, a tab, a carriage return, a newline or a form feed. A **digit** (**\d**) is any numeric character.

Figure 16.18 uses character classes in regular expressions. For this example, we use method DisplayMatches (lines 53–59) to display all matches to a regular expression.

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. 16.17 | Character classes.

Method DisplayMatches takes two strings representing the string to search and the regular expression to match. The method uses a foreach statement to display each Match in the MatchCollection object returned by the static method Matches of class Regex.

```
1  // Fig. 16.18: CharacterClasses.cs
2  // Demonstrate using character classes and quantifiers.
3  using System;
4  using System.Text.RegularExpressions;
5
```

Fig. 16.18 | Demonstrating using character classes and quantifiers. (Part 1 of 3.)

```
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
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]" );
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
```

Fig. 16.18 | Demonstrating using character classes and quantifiers. (Part 2 of 3.)

```
58         Console.WriteLine(); // move to the next line
59      } // end method DisplayMatches
60   } // end class CharacterClasses
```

```
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
```

Fig. 16.18 | Demonstrating using character classes and quantifiers. (Part 3 of 3.)

The first regular expression (line 15) matches digits in the `testString`. We use the digit character class (`\d`) to match any digit (0–9). We precede the regular expression `string` with `@`. Recall that backslashes within the double quotation marks following the `@` character are regular backslash characters, not the beginning of escape sequences. To define the regular expression without prefixing `@` to the string, you would need to escape every backslash character, as in

```
"\\d"
```

which makes the regular expression more difficult to read.

The output shows that the regular expression matches `1`, `2`, and `3` in the `testString`. You can also match anything that *isn't* a member of a particular character class using an uppercase instead of a lowercase letter. For example, the regular expression `"\D"` (line 19) matches any character that isn't a digit. Notice in the output that this includes punctuation and whitespace. Negating a character class matches *everything* that *isn't* a member of the character class.

The next regular expression (line 23) uses the character class `\w` to match any word character in the `testString`. Notice that each match consists of a single character. It would be useful to match a sequence of word characters rather than a single character. The

regular expression in line 28 uses the + quantifier to match a sequence of word characters. The **+ quantifier** matches one or more occurrences of the pattern to its left. There are three matches for this expression, each three characters long. 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. The regular expression `"\w+?"` (line 33) uses a lazy + quantifier to match the shortest sequence of word characters possible. This produces nine matches of length one instead of three matches of length three. Figure 16.19 lists other quantifiers that you can place after a pattern in a regular expression, and the purpose of each.

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. 16.19 | Quantifiers used in regular expressions.

Regular expressions are not limited to the character classes in Fig. 16.17. You can create your own character class by listing the members of the character class between square brackets, `[` and `]`. [*Note:* Metacharacters in square brackets are treated as literal characters.] You can include a range of characters using the `"-"` character. The regular expression in line 37 of Fig. 16.18 creates a character class to match any lowercase letter from `a` to `f`. These custom character classes match a single character that's a member of the class. The output shows three matches, `a`, `b` and `c`. Notice that `D`, `E` and `F` don't match the character class `[a-f]` because they're uppercase. You can negate a custom character class by placing a `"^"` character after the opening square bracket. The regular expression in line 41 matches any character that *isn't* in the range `a-f`. As with the predefined character classes, negating a custom character class matches *everything* that isn't a member, including punctuation and whitespace. You can also use quantifiers with custom character classes. The regular expression in line 45 uses a character class with two ranges of characters, `a-z` and `A-Z`, and the + quantifier to match a sequence of lowercase or uppercase letters. You can also use the `"."` (dot) character to match any character other than a newline. The regular expression `".*"` (line 49) matches any sequence of characters. The * quantifier matches zero or more occurrences of the pattern to its left. Unlike the + quantifier, the * quantifier can be used to match an empty string.

16.15.2 Complex Regular Expressions

The program of Fig. 16.20 tries to match birthdays to a regular expression. For demonstration purposes, the expression matches only birthdays that do not occur in April and

that belong to people whose names begin with "J". We can do this by combining the basic regular-expression techniques we've already discussed.

```
1  // Fig. 16.20: 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
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
```

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

Fig. 16.20 | A more complex regular expression.

Line 11 creates a `Regex` object and passes a regular-expression pattern `string` to its constructor. The first character in the regular expression, `"J"`, is a literal character. Any `string` matching this regular expression must start with `"J"`. The next part of the regular expression (`".*"`) matches any number of unspecified characters except newlines. The pattern `"J.*"` matches a person's name that starts with J and any characters that may come after that.

Next we match the person's birthday. We use the `\d` character class to match the first digit of the month. Since the birthday must not occur in April, the second digit in the month can't be `4`. We could use the character class `"[0-35-9]"` to match any digit other than `4`. However, .NET regular expressions allow you to subtract members from a character class, called **character-class subtraction**. In line 11, we use the pattern `"[\d-[4]]"` to match any digit other than `4`. When the `"-"` character in a character class is followed by a character class instead of a literal character, the `"-"` is interpreted as subtraction instead of a range of characters. 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 (`[4]`) must be the last item in the enclosing brackets (`[\d-[4]]`). This notation allows you to write shorter, easier-to-read regular expressions.

Although the "-" character indicates a range or character-class subtraction when it's enclosed in square brackets, instances of the "-" character outside a character class are treated as literal characters. Thus, the regular expression in line 11 searches for a `string` that starts with the letter "J", followed by any number of characters, followed by a two-digit number (of which the second digit cannot be 4), followed by a dash, another two-digit number, a dash and another two-digit number.

Lines 20–21 use a `foreach` statement to iterate through the `MatchCollection` object returned by method `Matches`, which received `testString` as an argument. For each `Match`, line 21 outputs the text that matched the regular expression. The output in Fig. 16.20 displays the two matches that were found in `testString`. Notice that both matches conform to the pattern specified by the regular expression.

16.15.3 Validating User Input with Regular Expressions and LINQ

The application in Fig. 16.21 presents a more involved example that uses regular expressions to validate name, address and telephone-number information input by a user.

```
1  // Fig. 16.21: 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
17       // handles OK Button's Click event
18       private void okButton_Click( object sender, EventArgs e )
19       {
20          // find blank TextBoxes and order by TabIndex
21          var emptyBoxes =
22             from Control currentControl in Controls
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          {
```

Fig. 16.21 | Validating user information using regular expressions. (Part 1 of 4.)

```
32             // display message box indicating missing information
33             MessageBox.Show( "Please fill in all fields",
34                "Missing Information", MessageBoxButtons.OK,
35                MessageBoxIcon.Error );
36
37             emptyBoxes.First().Select(); // select first empty TextBox
38          } // end if
39          else
40          {
41             // check for invalid input
42             if ( !ValidateInput( lastNameTextBox.Text,
43                "^[A-Z][a-zA-Z]*$", "Invalid last name" ) )
44
45                lastNameTextBox.Select(); // select invalid TextBox
46             else if ( !ValidateInput( firstNameTextBox.Text,
47                "^[A-Z][a-zA-Z]*$", "Invalid first name" ) )
48
49                firstNameTextBox.Select(); // select invalid TextBox
50             else if ( !ValidateInput( addressTextBox.Text,
51                @"^[0-9]+\s+([a-zA-Z]+|[a-zA-Z]+\s[a-zA-Z]+)$",
52                "Invalid address" ) )
53
54                addressTextBox.Select(); // select invalid TextBox
55             else if ( !ValidateInput( cityTextBox.Text,
56                @"^([a-zA-Z]+|[a-zA-Z]+\s[a-zA-Z]+)$", "Invalid city" ) )
57
58                cityTextBox.Select(); // select invalid TextBox
59             else if ( !ValidateInput( stateTextBox.Text,
60                @"^([a-zA-Z]+|[a-zA-Z]+\s[a-zA-Z]+)$", "Invalid state" ) )
61
62                stateTextBox.Select(); // select invalid TextBox
63             else if ( !ValidateInput( zipCodeTextBox.Text,
64                @"^\d{5}$", "Invalid zip code" ) )
65
66                zipCodeTextBox.Select(); // select invalid TextBox
67             else if ( !ValidateInput( phoneTextBox.Text,
68                @"^[1-9]\d{2}-[1-9]\d{2}-\d{4}$",
69                "Invalid phone number" ) )
70
71                phoneTextBox.Select(); // 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                   MessageBoxButtons.OK, MessageBoxIcon.Information );
77                Application.Exit(); // exit the application
78             } // end else
79          } // end else
80       } // end method okButton_Click
81
```

Fig. 16.21 | Validating user information using regular expressions. (Part 2 of 4.)

```
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                  MessageBoxButtons.OK, MessageBoxIcon.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
```

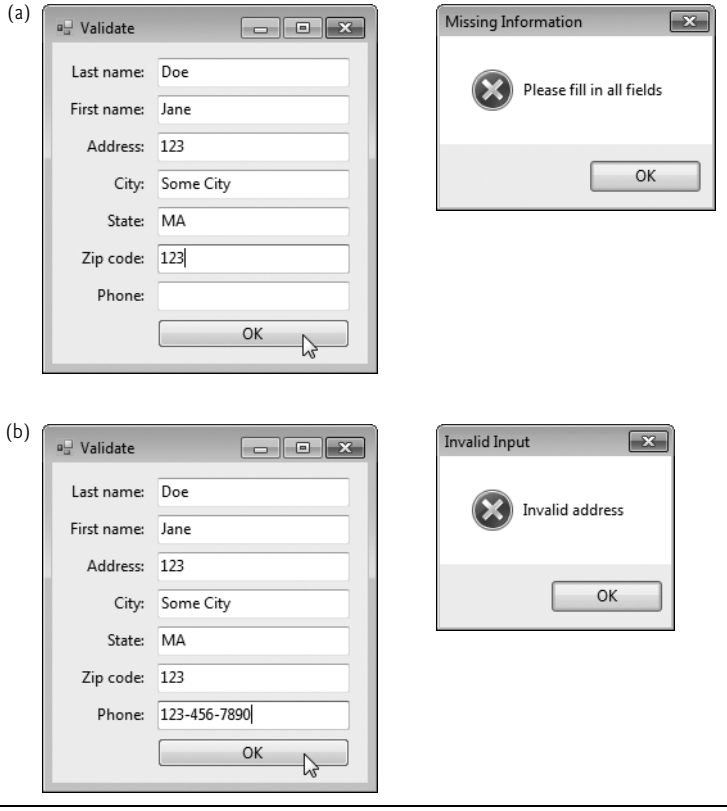

Fig. 16.21 | Validating user information using regular expressions. (Part 3 of 4.)

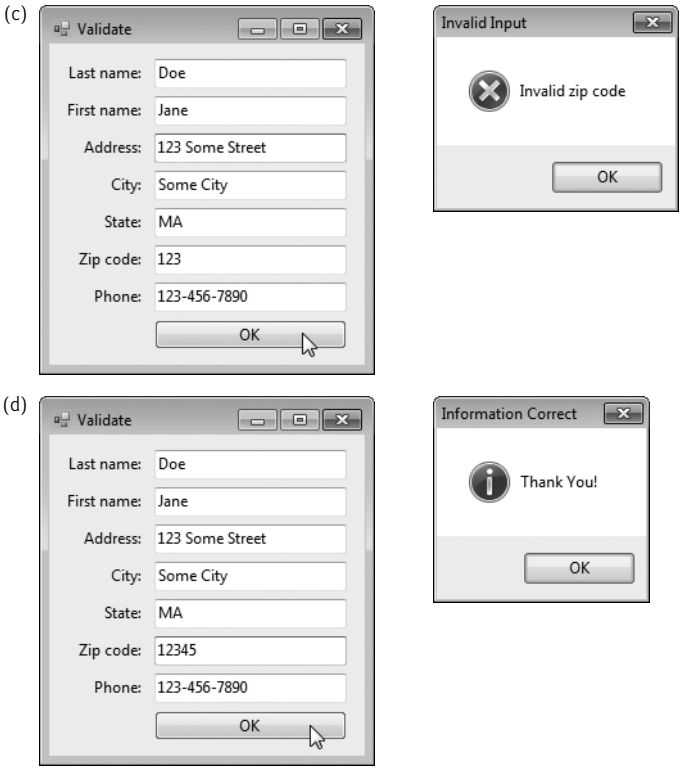

Fig. 16.21 | Validating user information using regular expressions. (Part 4 of 4.)

When a user clicks **OK**, the program uses a LINQ query to select any empty `TextBox`es (lines 22–27) from the `Controls` collection. Notice that we explicitly declare the type of the range variable in the `from` clause (line 22). When working with nongeneric collections, such as `Controls`, you must explicitly type the range variable. The first `where` clause (line 23) determines whether the `currentControl` is a `TextBox`. The **`let`** clause (line 24) creates and initializes a variable in a LINQ query for use later in the query. Here, we use the `let` clause to define variable `box` as a `TextBox`, which contains the `Control` object cast to a `TextBox`. This allows us to use the control in the LINQ query as a `TextBox`, enabling access to its properties (such as `Text`). You may include a second `where` clause after the `let` clause. The second `where` clause determines whether the `TextBox`'s `Text` property is empty. If one or more `TextBox`es are empty (line 30), the program displays a message to the user (lines 33–35) that all fields must be filled in before the program can validate the information. Line 37 calls the `Select` method of the first `TextBox` in the query result so that the user can begin typing in that `TextBox`. The query sorted the `TextBox`es by `TabIndex` (line 26) so the first `TextBox` in the query result is the first empty `TextBox` on the `Form`. If there are no empty fields, lines 39–71 validate the user input.

We call method `ValidateInput` to determine whether the user input matches the specified regular expressions. `ValidateInput` (lines 83–98) takes as arguments the text input by the user (`input`), the regular expression the input must match (`expression`) and

a message to display if the input is invalid (`message`). Line 87 calls `Regex` `static` method `Match`, passing both the `string` to validate and the regular expression as arguments. The **`Success`** property of class `Match` indicates whether method `Match`'s first argument matched the pattern specified by the regular expression in the second argument. If the value of `Success` is `false` (i.e., there was no match), lines 93–94 display the error message passed as an argument to method `ValidateInput`. Line 97 then returns the value of the `Success` property. If `ValidateInput` returns `false`, the `TextBox` containing invalid data is selected so the user can correct the input. If all input is valid—the `else` statement (lines 72–78) displays a message dialog stating that all input is valid, and the program terminates when the user dismisses the dialog.

In the previous example, we searched a `string` for substrings that matched a regular expression. In this example, we want to ensure that the entire `string` for each input conforms to a particular regular expression. For example, we want to accept `"Smith"` as a last name, but not `"9@Smith#"`. In a regular expression that begins with a `"^"` character and ends with a `"$"` character (e.g., line 43), the characters `"^"` and `"$"` 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 regular expressions in lines 43 and 47 use a character class to match an uppercase first letter followed by letters of any case—`a-z` matches any lowercase letter, and `A-Z` matches any uppercase letter. The `*` quantifier signifies that the second range of characters may occur zero or more times in the `string`. Thus, this expression matches any `string` consisting of one uppercase letter, followed by zero or more additional letters.

The `\s` character class matches a single whitespace character (lines 51, 56 and 60). In the expression `"\d{5}"`, used for the `zipCode` `string` (line 64), `{5}` is a quantifier (see Fig. 16.19). The pattern to the left of `{n}` must occur exactly `n` times. Thus `"\d{5}"` matches any five digits. Recall that the character `"|"` (lines 51, 56 and 60) matches the expression to its left *or* the expression to its right. In line 51, we use the character `"|"` to indicate that 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. Note the use of parentheses to group parts of the regular expression. This ensures that `"|"` is applied to the correct parts of the pattern.

The **Last Name:** and **First Name:** `TextBox`es each accept `string`s that begin with an uppercase letter (lines 43 and 47). The regular expression for the **Address:** `TextBox` (line 51) matches a number of at least one digit, followed by a space and then either one or more letters or else one or more letters followed by a space and another series of one or more letters. Therefore, `"10 Broadway"` and `"10 Main Street"` are both valid addresses. As currently formed, the regular expression in line 51 doesn't match an address that does not start with a number, or that has more than two words. The regular expressions for the **City:** (line 56) and **State:** (line 60) `TextBox`es match any word of at least one character or, alternatively, any two words of at least one character if the words are separated by a single space. This means both `Waltham` and `West Newton` would match. Again, these regular expressions would not accept names that have more than two words. The regular expression for the **Zip code:** `TextBox` (line 64) ensures that the zip code is a five-digit number. The regular expression for the **Phone:** `TextBox` (line 68) indicates that the phone number must be of the form `xxx-yyy-yyyy`, where the `x`s represent the area code and the `y`s the number. The first `x` and the first `y` cannot be zero, as specified by the range `[1-9]` in each case.

16.15.4 Regex Methods Replace and Split

Sometimes it's useful to replace parts of one `string` with another or to split a `string` according to a regular expression. For this purpose, class `Regex` provides `static` and instance versions of methods `Replace` and `Split`, which are demonstrated in Fig. 16.22.

```
1  // Fig. 16.22: 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
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
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
```

Fig. 16.22 | Using `Regex` methods `Replace` and `Split`. (Part 1 of 2.)

```
45         // delete ", " at the end of output string
46         Console.WriteLine( output.Substring( 0, output.Length - 2 ) + "]" );
47      } // end Main
48   } // end class RegexSubstitution
```

```
First test string: This sentence ends in 5 stars *****
^ substituted for *: This sentence ends in 5 stars ^^^^^
"carets" substituted for "stars": This sentence ends in 5 carets ^^^^^
Every word replaced by "word": word word word word word word ^^^^^

Second test string: 1, 2, 3, 4, 5, 6, 7, 8
Replace first 3 digits by "digit": digit, digit, digit, 4, 5, 6, 7, 8
String split at commas ["1", "2", "3", "4", "5", "6", "7", "8"]
```

Fig. 16.22 | Using `Regex` methods `Replace` and `Split`. (Part 2 of 2.)

`Regex` method **`Replace`** replaces text in a `string` with new text wherever the original `string` matches a regular expression. We use two versions of this method in Fig. 16.22. The first version (line 18) is a `static` method and takes three parameters—the `string` to modify, the `string` containing the regular expression to match and the replacement `string`. Here, `Replace` replaces every instance of `"*"` in `testString1` with `"^"`. Notice that the regular expression (`"\*"`) precedes character `*` with a backslash (`\`). Normally, `*` is a quantifier indicating that a regular expression should match any number of occurrences of a preceding pattern. However, in line 18, we want to find all occurrences of the literal character `*`; to do this, we must escape character `*` with character `\`. By escaping a special regular-expression character, we tell the regular-expression matching engine to find the actual character `*` rather than use it as a quantifier.

The second version of method `Replace` (line 34) is an instance method that uses the regular expression passed to the constructor for `testRegex1` (line 12) to perform the replacement operation. Line 12 instantiates `testRegex1` with argument `@"\d"`. The call to instance method `Replace` in line 34 takes three arguments—a `string` to modify, a `string` containing the replacement text and an integer specifying the number of replacements to make. In this case, line 34 replaces the first three instances of a digit (`"\d"`) in `testString2` with the text `"digit"`.

Method **`Split`** divides a `string` into several substrings. The original `string` is broken at delimiters that match a specified regular expression. Method `Split` returns an `array` containing the substrings. In line 39, we use `static` method `Split` to separate a `string` of comma-separated integers. The first argument is the `string` to split; the second argument is the regular expression that represents the delimiter. The regular expression `",\s"` separates the substrings at each comma. By matching a whitespace character (`\s` in the regular expression), we eliminate the extra spaces from the resulting substrings.