5.10 Char Methods

The simple types are actually aliases for struct types. For instance, an int is defined by struct System.Int32, a long by System.Int64 and so on. All struct types derive from class ValueType, which derives from object. In the struct Char, which is the struct for characters, take at least one character argument and perform either a test or a manipulation on the character. We present several of these methods in the next example. Figure 5.9 demonstrates static methods that test characters to determine whether they're of a specific character type and static methods that perform case conversions on characters.

```
1     // Fig. 5.9 : StaticCharMethods.cs
2     // Demonstrates static character-testing and case-conversion methods
3     // from Char struct
4     using System;
5
6     class StaticCharMethods
7     {
8        static void Main( string[] args )
9        {
10          Console.Write( "Enter a character: " );
11          char character = Convert.ToChar( Console.ReadLine() );
12
13          Console.WriteLine( "is digit: {0}", Char.IsDigit( character ) );
14          Console.WriteLine( "is letter: {0}", Char.IsLetter( character ) );
15          Console.WriteLine( "is letter or digit: {0}",
16             Char.IsLetterOrDigit( character ) );
17          Console.WriteLine( "is lower case: {0}",
18             Char.IsLower( character ) );
19          Console.WriteLine( "is upper case: {0}",
20             Char.IsUpper( character ) );
21          Console.WriteLine( "to upper case: {0}",
22             Char.ToUpper( character ) );
23          Console.WriteLine( "to lower case: {0}",
24             Char.ToLower( character ) );
25          Console.WriteLine( "is punctuation: {0}",
26             Char.IsPunctuation( character ) );
27          Console.WriteLine( "is symbol: {0}", Char.IsSymbol( character ) );
28       } // end Main
29    } // end class StaticCharMethods
```

```
Enter a character: A
is digit: False
is letter: True
is letter or digit: True
is lower case: False
is upper case: True
to upper case: A
to lower case: a
is punctuation: False
is symbol: False
```

Fig. 5.9| Char's static character-testing and case-conversion methods. (Part 1 of 2.)

```
Enter a character: 8
is digit: True
is letter: False
is letter or digit: True
is lower case: False
is upper case: False
to upper case: 8
to lower case: 8
is punctuation: False
is symbol: False
```

```
Enter a character: @
is digit: False
is letter: False
is letter or digit: False
is lower case: False
is upper case: False
to upper case: @
to lower case: @
is punctuation: True
is symbol: False
```

```
Enter a character: m
is digit: False
is letter: True
is letter or digit: True
is lower case: True
is upper case: False
to upper case: M
to lower case: m
is punctuation: False
is symbol: False
```

Fig. 5.9 | `Char`'s `static` character-testing and case-conversion methods. (Part 2 of 2.)

After the user enters a character, lines 13–27 analyze it. Line 13 uses Char method IsDigit to determine whether character is defined as a digit. If so, the method returns true; otherwise, it returns false (note again that bool values are output capitalized). Line 14 uses Char method IsLetter to determine whether character character is a letter. Line 16 uses Char method IsLetterOrDigit to determine whether character character is a letter or a digit. Line 18 uses Char method IsLower to determine whether character character is a lowercase letter. Line 20 uses Char method IsUpper to determine whether character character is an uppercase letter. Line 22 uses Char method ToUpper to convert character character to its uppercase equivalent. The method returns the converted character if the character has an uppercase equivalent; otherwise, the method returns its original argument.

Line 24 uses Char method ToLower to convert character character to its lowercase equivalent. The method returns the converted character if the character has a lowercase equivalent; otherwise, the method returns its original argument. Line 26 uses Char method IsPunctuation to determine whether character is a punctuation mark, such as "!", ":" or ")". Line 27 uses Char method IsSymbol to determine whether character character is a symbol, such as "+", "=" or "^". Structure type Char also contains other methods not shown in this example. Many of the static methods are similar—for instance, IsWhiteSpace is used to determine whether a certain character is a whitespace character (e.g., newline, tab or space). The struct also contains several public instance methods; many of these, such as methods ToString and Equals, are methods that we have seen before in other classes. This group includes method CompareTo, which is used to compare two character values with one another.

5.11 Regular Expressions

We now introduce 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.

5.11.1 Simple Regular Expressions and Class Regex

The .NET Framework provides several classes to help developers manipulate regular expressions. Figure 5.10demonstrates 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 Matches. 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. 5.16: 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: " );
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. 5.10 | Demonstrating basic regular expressions.

Regular expressions can also be used to match a sequence of literal characters any- where 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 (line27) 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. 5.11 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.

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. 5.11 | Character classes.

Figure 5.12uses character classes in regular expressions. For this example, we use method DisplayMatches (lines 53–59) to display all matches to a regular expression. Method DisplayMatches takes two strings representing the string to search and the reg- ular 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. 5.18: 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
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
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. 5.12 | Demonstrating using character classes and quantifiers.

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

Regular expressions are not limited to the character classes in Fig. 5.17. You can create your own character class by listing the members of the character class between square brackets, [and]. You can include a range of characters using the "-" character. The regular expression in line 37 of Fig. 5.12 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.

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. 5.13 | Quantifiers used in regular expressions.

The regular expression in line 41 matches any character that isn't in the range a-f. As with the predefined char- acter 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.*

5.11.2 Complex Regular Expressions

The program of Fig. 5.14 tries to match birthdays to a regular expression. For demon- stration 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. 5.20: 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. 5.14 | 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. 5.14 displays the two matches that were found in testString. Notice that both matches conform to the pattern specified by the regular expression.

5.11.4 Regex Methods Replace and Split

Sometimes it's useful to replace parts of one string with another or to split a string ac- cording to a regular expression. For this purpose, class Regex provides static and instance versions of methods Replace and Split, which are demonstrated in Fig. 5.15 .

```
1  // Fig. 5.15 : 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
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. 5.15 | Using `Regex` methods `Replace` and `Split`.