Chapter 25. Regular Expressions

The regular expressions language identifies character patterns. The .NET types supporting regular expressions are based on Perl 5 regular expressions and support both search and search/replace functionality.

Regular expressions are used for tasks such as:

- Validating text input such as passwords and phone numbers
- Parsing textual data into more structured forms (e.g., a NuGet version string)
- Replacing patterns of text in a document (e.g., whole words only)

This chapter is split into both conceptual sections teaching the basics of regular expressions in .NET, and reference sections describing the regular expressions language.

All regular expression types are defined in `System.Text.RegularExpressions`.

> **NOTE**
>
> The samples in this chapter are all preloaded into LINQPad, which also includes an interactive RegEx tool (press Ctrl+Shift+F1). An online tool is available at *http://regexstorm.net/tester*.

Regular Expression Basics

One of the most common regular expression operators is a *quantifier*. `?` is a quantifier that matches the preceding item 0 or 1 time. In other words, `?` means *optional*. An item is either a single character or a complex structure

of characters in square brackets. For example, the regular expression `"colou?r"` matches `color` and `colour`, but not `colouur`:

```
Console.WriteLine (Regex.Match ("color",   @"colou?r").Success);  // True
Console.WriteLine (Regex.Match ("colour",  @"colou?r").Success);  // True
Console.WriteLine (Regex.Match ("colouur", @"colou?r").Success);  // False
```

`Regex.Match` searches within a larger string. The object that it returns has properties for the `Index` and `Length` of the match as well as the actual `Value` matched:

```
Match m = Regex.Match ("any colour you like", @"colou?r");

Console.WriteLine (m.Success);     // True
Console.WriteLine (m.Index);       // 4
Console.WriteLine (m.Length);      // 6
Console.WriteLine (m.Value);       // colour
Console.WriteLine (m.ToString());  // colour
```

You can think of `Regex.Match` as a more powerful version of the `string`'s `IndexOf` method. The difference is that it searches for a *pattern* rather than a literal string.

The `IsMatch` method is a shortcut for calling `Match` and then testing the `Success` property.

The regular expressions engine works from left to right by default, so only the leftmost match is returned. You can use the `NextMatch` method to return more matches:

```
Match m1 = Regex.Match ("One color? There are two colours in my head!",
                        @"colou?rs?");
Match m2 = m1.NextMatch();
Console.WriteLine (m1);         // color
Console.WriteLine (m2);         // colours
```

The `Matches` method returns all matches in an array. We can rewrite the preceding example, as follows:

```
foreach (Match m in Regex.Matches
          ("One color? There are two colours in my head!", @"colou?rs?"))
  Console.WriteLine (m);
```

Another common regular expressions operator is the *alternator*, expressed with a vertical bar, `|`. An alternator expresses alternatives. The following matches “Jen”, “Jenny”, and “Jennifer”:

```
Console.WriteLine (Regex.IsMatch ("Jenny", "Jen(ny|nifer)?"));  // True
```

The brackets around an alternator separate the alternatives from the rest of the expression.

> **NOTE**
>
> You can specify a timeout when matching regular expressions. If a match operation takes longer than the specified `TimeSpan`, a `RegexMatchTimeoutException` is thrown. This can be useful if your program processes user-supplied regular expressions because it prevents malformed regular expressions from infinitely spinning.

Compiled Regular Expressions

In some of the preceding examples, we called a static `RegEx` method repeatedly with the same pattern. An alternative approach in these cases is to instantiate a `Regex` object with the pattern and `RegexOptions.Compiled` and then call instance methods:

```
Regex r = new Regex (@"sausages?", RegexOptions.Compiled);
Console.WriteLine (r.Match ("sausage"));   // sausage
Console.WriteLine (r.Match ("sausages"));  // sausages
```

`RegexOptions.Compiled` instructs the `RegEx` instance to use lightweight code generation (`DynamicMethod` in `Reflection.Emit`) to dynamically build and compile code tailored to that particular regular expression. This results in faster matching, at the expense of an initial compilation cost.

You can also instantiate a `Regex` object without using `RegexOptions.Compiled`. A `Regex` instance is immutable.

> **NOTE**
>
> The regular expressions engine is fast. Even without compilation, a simple match typically takes less than a microsecond.

RegexOptions

The `RegexOptions` flags enum lets you tweak matching behavior. A common use for `RegexOptions` is to perform a case-insensitive search:

```
Console.WriteLine (Regex.Match ("a", "A", RegexOptions.IgnoreCase)); // a
```

This applies the current culture's rules for case equivalence. The `CultureInvariant` flag lets you request the invariant culture instead:

```
Console.WriteLine (Regex.Match ("a", "A", RegexOptions.IgnoreCase
                                        | RegexOptions.CultureInvariant));
```

You can activate most of the `RegexOptions` flags within a regular expression itself, using a single-letter code, as follows:

```
Console.WriteLine (Regex.Match ("a", @"(?i)A"));                    // a
```

You can turn options on and off throughout an expression:

```
Console.WriteLine (Regex.Match ("AAAa", @"(?i)a(?-i)a"));           // Aa
```

Another useful option is `IgnorePatternWhitespace` or `(?x)`. This allows you to insert whitespace to make a regular expression more readable—without the whitespace being taken literally.

The `NonBacktracking` option (from .NET 7) instructs the regex engine to use a forwards-only matching algorithm. This usually results in slower performance and disables some advanced features such as lookahead or lookbehind. However, it also prevents malformed or maliciously constructed expressions from taking near-infinite time, mitigating a potential denial-of-service attack when processing user-supplied regular expressions (a *ReDOS* attack). Specifying a timeout is also useful in this scenario.

Table 25-1 lists all `RegExOptions` values along with their single-letter codes.

Table 25-1. Regular expression options

Enum value	Regular expressions code	Description
`None`		
`IgnoreCase`	`i`	Ignores case (by default, regular expressions are case sensitive)
`Multiline`	`m`	Changes `^` and `$` so that they match the start/end of a line instead of start/end of the string
`ExplicitCapture`	`n`	Captures only explicitly named or explicitly numbered groups (see "Groups")
`Compiled`		Forces compilation to IL (see "Compiled Regular Expressions")
`Singleline`	`s`	Makes `.` match every character (instead of matching every character except `\n`)
`IgnorePatternWhitespace`	`x`	Eliminates unescaped whitespace from the pattern
`RightToLeft`	`r`	Searches from right to left; can't be specified midstream
`ECMAScript`		Forces ECMA compliance (by default, the implementation is not ECMA compliant)

Enum value	Regular expressions code	Description
`CultureInvariant`		Turns off culture-specific behavior for string comparisons
`NonBacktracking`		Disables backtracking to ensure predictable (albeit slower) performance

Character Escapes

Regular expressions have the following metacharacters, which have a special rather than literal meaning:

\ * + ? | { [() ^ $. #

To use a metacharacter literally, you must prefix, or *escape*, the character with a backslash. In the following example, we escape the `?` character to match the string `"what?"`:

```
Console.WriteLine (Regex.Match ("what?", @"what\?")); // what? (correct)
Console.WriteLine (Regex.Match ("what?", @"what?"));  // what  (incorrect)
```

> **NOTE**
>
> If the character is inside a *set* (square brackets), this rule does not apply, and the metacharacters are interpreted literally. We discuss sets in the following section.

The `Regex`'s `Escape` and `Unescape` methods convert a string containing regular expression metacharacters by replacing them with escaped equivalents, and vice versa:

```
Console.WriteLine (Regex.Escape   (@"?"));     // \?
Console.WriteLine (Regex.Unescape (@"\?"));    // ?>
```

All the regular expression strings in this chapter are expressed with the C# `@` literal. This is to bypass C#'s escape mechanism, which also uses the backslash. Without the `@`, a literal backslash would require four backslashes:

```
Console.WriteLine (Regex.Match ("\\", "\\\\"));    // \
```

Unless you include the `(?x)` option, spaces are treated literally in regular expressions:

```
Console.Write (Regex.IsMatch ("hello world", @"hello world"));  // True
```

Character Sets

Character sets act as wildcards for a particular set of characters.

Expression	Meaning	Inverse ("not")
`[abcdef]`	Matches a single character in the list.	`[^abcdef]`
`[a-f]`	Matches a single character in a *range*.	`[^a-f]`
`\d`	Matches anything in the Unicode *digits* category. In ECMAScript mode, `[0-9]`.	`\D`
`\w`	Matches a *word* character (by default, varies according to `CultureInfo.CurrentCulture`; for example, in English, same as `[a-zA-Z_0-9]`).	`\W`
`\s`	Matches a whitespace character; that is, anything for which `char.IsWhiteSpace` returns true (including Unicode spaces). In ECMAScript mode, `[\n\r\t \f\v ]`.	`\S`
`\p{`*category*`}`	Matches a character in a specified *category*.	`\P`
.	(Default mode) Matches any character except `\n`.	`\n`
.	(`SingleLine` mode) Matches any character.	`\n`

To match exactly one of a set of characters, put the character set in square brackets:

```
Console.Write (Regex.Matches ("That is that.", "[Tt]hat").Count);   // 2
```

To match any character *except* those in a set, put the set in square brackets with a `^` symbol before the first character:

```
Console.Write (Regex.Match ("quiz qwerty", "q[^aeiou]").Index);    // 5
```

You can specify a range of characters by using a hyphen. The following regular expression matches a chess move:

```
Console.Write (Regex.Match ("b1-c4", @"[a-h]\d-[a-h]\d").Success);  // True
```

`\d` indicates a digit character, so `\d` will match any digit. `\D` matches any nondigit character.

`\w` indicates a word character, which includes letters, numbers, and the underscore. `\W` matches any nonword character. These work as expected for non-English letters, too, such as Cyrillic.

`.` matches any character except `\n` (but allows `\r`).

`\p` matches a character in a specified category, such as `{Lu}` for uppercase letter or `{P}` for punctuation (we list the categories in the reference section later in the chapter):

```
Console.Write (Regex.IsMatch ("Yes, please", @"\p{P}"));   // True
```

We will find more uses for `\d`, `\w`, and `.` when we combine them with *quantifiers*.

Quantifiers

Quantifiers match an item a specified number of times.

Quantifier	Meaning
`*`	Zero or more matches
`+`	One or more matches
`?`	Zero or one match
`{n}`	Exactly *n* matches
`{n,}`	At least *n* matches
`{n,m}`	Between *n* and *m* matches

The * quantifier matches the preceding character or group zero or more times. The following matches *cv.docx*, along with any numbered versions of the same file (e.g., *cv2.docx*, *cv15.docx*):

```
Console.Write (Regex.Match ("cv15.docx", @"cv\d*\.docx").Success);  // True
```

Notice that we must escape the period in the file extension using a backslash.

The following allows anything between *cv* and *.docx* and is equivalent to `dir cv*.docx`:

```
Console.Write (Regex.Match ("cvjoint.docx", @"cv.*\.docx").Success); // True
```

The + quantifier matches the preceding character or group one or more times. For example:

```
Console.Write (Regex.Matches ("slow! yeah slooow!", "slo+w").Count);  // 2
```

The {} quantifier matches a specified number (or range) of repetitions. The following matches a blood pressure reading:

```
Regex bp = new Regex (@"\d{2,3}/\d{2,3}");
Console.WriteLine (bp.Match ("It used to be 160/110"));  // 160/110
Console.WriteLine (bp.Match ("Now it's only 115/75"));   // 115/75
```

Greedy Versus Lazy Quantifiers

By default, quantifiers are *greedy*, as opposed to *lazy*. A greedy quantifier repeats as *many* times as it can before advancing. A lazy quantifier repeats as *few* times as it can before advancing. You can make any quantifier lazy by suffixing it with the ? symbol. To illustrate the difference, consider the following HTML fragment:

```
string html = "<i>By default</i> quantifiers are <i>greedy</i> creatures";
```

Suppose that we want to extract the two phrases in italics. If we execute the following

```
foreach (Match m in Regex.Matches (html, @"<i>.*</i>"))
  Console.WriteLine (m);
```

the result is not two matches, but a *single* match:

```
<i>By default</i> quantifiers are <i>greedy</i>
```

The problem is that our * quantifier greedily repeats as many times as it can before matching </i>. So, it passes right by the first </i>, stopping only at the final </i> (the *last point* at which the rest of the expression can still match).

If we make the quantifier lazy, the * bails out at the *first* point at which the rest of the expression can match:

```
foreach (Match m in Regex.Matches (html, @"<i>.*?</i>"))
  Console.WriteLine (m);
```

Here's the result:

```
<i>By default</i>
<i>greedy</i>
```

Zero-Width Assertions

The regular expressions language lets you place conditions on what should occur *before* or *after* a match, through *lookbehind*, *lookahead*, *anchors*, and *word boundaries*. These are called *zero-width* assertions because they don't increase the width (or length) of the match itself.

Lookahead and Lookbehind

The `(?=expr)` construct checks whether the text that follows matches *`expr`*, without including `expr` in the result. This is called *positive lookahead*. In the following example, we look for a number followed by the word "miles":

```
Console.WriteLine (Regex.Match ("say 25 miles more", @"\d+\s(?=miles)"));

OUTPUT: 25
```

Notice that the word "miles" was not returned in the result, even though it was required to *satisfy* the match.

After a successful *lookahead*, matching continues as though the sneak preview never took place. So, if we append `.*` to our expression like this:

```
Console.WriteLine (Regex.Match ("say 25 miles more", @"\d+\s(?=miles).*"));
```

the result is `25 miles more`.

Lookahead can be useful in enforcing rules for a strong password. Suppose that a password must be at least six characters and contain at least one digit. With a lookup, we could achieve this, as follows:

```
string password = "...";
bool ok = Regex.IsMatch (password, @"(?=.*\d).{6,}");
```

This first performs a *lookahead* to ensure that a digit occurs somewhere in the string. If satisfied, it returns to its position before the sneak preview began and matches six or more characters. (In "Cookbook Regular Expressions", we include a more substantial password validation example.)

The opposite is the *negative lookahead* construct, `(?!expr)`. This requires that the match *not* be followed by `expr`. The following expression matches "good"—unless "however" or "but" appears later in the string:

```
string regex = "(?i)good(?!.*(however|but))";
Console.WriteLine (Regex.IsMatch ("Good work! But...",  regex));  // False
Console.WriteLine (Regex.IsMatch ("Good work! Thanks!", regex));  // True
```

The `(?<=expr)` construct denotes *positive lookbehind* and requires that a match be *preceded* by a specified expression. The opposite construct, `(?<!expr)`, denotes *negative lookbehind* and requires that a match *not be preceded* by a specified expression. For example, the following matches "good"—unless "however" appears *earlier* in the string:

```
string regex = "(?i)(?<!however.*)good";
Console.WriteLine (Regex.IsMatch ("However good, we...", regex)); // False
Console.WriteLine (Regex.IsMatch ("Very good, thanks!", regex));  // True
```

We could improve these examples by adding *word boundary assertions*, which we introduce shortly.

Anchors

The anchors `^` and `$` match a particular *position*. By default:

`^`

Matches the *start* of the string

`$`

Matches the *end* of the string

> **NOTE**
>
> `^` has two context-dependent meanings: an *anchor* and a *character class negator*.
>
> `$` has two context-dependent meanings: an *anchor* and a *replacement group denoter*.

For example:

```
Console.WriteLine (Regex.Match ("Not now", "^[Nn]o"));   // No
Console.WriteLine (Regex.Match ("f = 0.2F", "[Ff]$"));   // F
```

When you specify `RegexOptions.Multiline` or include `(?m)` in the expression:

- `^` matches the start of the string or *line* (directly after a `\n`).
- `$` matches the end of the string or *line* (directly before a `\n`).

There's a catch to using `$` in multiline mode: a new line in Windows is nearly always denoted with `\r\n` rather than just `\n`. This means that for `$` to be useful for Windows files, you must usually match the `\r`, as well, with a *positive lookahead*:

```
(?=\r?$)
```

The *positive lookahead* ensures that `\r` doesn't become part of the result. The following matches lines that end in `".txt"`:

```
string fileNames = "a.txt" + "\r\n" + "b.docx" + "\r\n" + "c.txt";
string r = @".+\.txt(?=\r?$)";
foreach (Match m in Regex.Matches (fileNames, r, RegexOptions.Multiline))
  Console.Write (m + " ");

OUTPUT: a.txt c.txt
```

The following matches all empty lines in string `s`:

```
MatchCollection emptyLines = Regex.Matches (s, "^(?=\r?$)",
                                            RegexOptions.Multiline);
```

The following matches all lines that are either empty or contain only whitespace:

```
MatchCollection blankLines = Regex.Matches (s, "^[ \t]*(?=\r?$)",
                                            RegexOptions.Multiline);
```

NOTE

Because an anchor matches a position rather than a character, specifying an anchor on its own matches an empty string:

```
Console.WriteLine (Regex.Match ("x", "$").Length);   // 0
```

Word Boundaries

The word boundary assertion `\b` matches where word characters (`\w`) adjoin either:

- Nonword characters (`\W`)
- The beginning/end of the string (`^` and `$`)

`\b` is often used to match whole words:

```
foreach (Match m in Regex.Matches ("Wedding in Sarajevo", @"\b\w+\b"))
  Console.WriteLine (m);

Wedding
in
Sarajevo
```

The following statements highlight the effect of a word boundary:

```
int one = Regex.Matches ("Wedding in Sarajevo", @"\bin\b").Count; // 1
int two = Regex.Matches ("Wedding in Sarajevo", @"in").Count;     // 2
```

The next query uses *positive lookahead* to return words followed by "(sic)":

```
string text = "Don't loose (sic) your cool";
Console.Write (Regex.Match (text, @"\b\w+\b\s(?=\(sic\))"));  // loose
```

Groups

Sometimes, it's useful to separate a regular expression into a series of subexpressions, or *groups*. For instance, consider the following regular expression that represents a US phone number such as 206-465-1918:

```
\d{3}-\d{3}-\d{4}
```

Suppose that we want to separate this into two groups: area code and local number. We can achieve this by using parentheses to *capture* each group:

```
(\d{3})-(\d{3}-\d{4})
```

We then retrieve the groups programmatically:

```
Match m = Regex.Match ("206-465-1918", @"(\d{3})-(\d{3}-\d{4})");

Console.WriteLine (m.Groups[1]);   // 206
Console.WriteLine (m.Groups[2]);   // 465-1918
```

The zeroth group represents the entire match. In other words, it has the same value as the match's `Value`:

```
Console.WriteLine (m.Groups[0]);   // 206-465-1918
Console.WriteLine (m);             // 206-465-1918
```

Groups are part of the regular expressions language itself. This means that you can refer to a group within a regular expression. The `\n` syntax lets you index the group by group number `n` within the expression. For example, the expression `(\w)ee\1` matches `deed` and `peep`. In the following example, we find all words in a string starting and ending in the same letter:

```
foreach (Match m in Regex.Matches ("pop pope peep", @"\b(\w)\w+\1\b"))
  Console.Write (m + " ");  // pop peep
```

The brackets around the `\w` instruct the regular expressions engine to store the submatch in a group (in this case, a single letter) so that it can be used later. We refer to that group later using `\1`, meaning the first group in the expression.

Named Groups

In a long or complex expression, it can be easier to work with groups by *name* rather than index. Here's a rewrite of the previous example, using a

group that we name `'letter'`:

```
string regEx =
  @"\b"             +  // word boundary
  @"(?'letter'\w)"  +  // match first letter, and name it 'letter'
  @"\w+"            +  // match middle letters
  @"\k'letter'"     +  // match last letter, denoted by 'letter'
  @"\b";               // word boundary

foreach (Match m in Regex.Matches ("bob pope peep", regEx))
  Console.Write (m + " ");  // bob peep
```

Here's how to name a captured group:

```
(?'group-name'group-expr)  or  (?<group-name>group-expr)
```

And here's how to refer to a group:

```
\k'group-name'  or  \k<group-name>
```

The following example matches a simple (non-nested) XML/HTML element by looking for start and end nodes with a matching name:

```
string regFind =
  @"<(?'tag'\w+?).*>" +  // lazy-match first tag, and name it 'tag'
  @"(?'text'.*?)"     +  // lazy-match text content, name it 'text'
  @"</\k'tag'>";         // match last tag, denoted by 'tag'

Match m = Regex.Match ("<h1>hello</h1>", regFind);
Console.WriteLine (m.Groups ["tag"]);          // h1
Console.WriteLine (m.Groups ["text"]);         // hello
```

Allowing for all possible variations in XML structure, such as nested elements, is more complex. The .NET regular expressions engine has a sophisticated extension called "matched balanced constructs" that can assist

with nested tags—information on this is available on the internet and in *Mastering Regular Expressions* (O'Reilly) by Jeffrey E. F. Friedl.

Replacing and Splitting Text

The `RegEx.Replace` method works like `string.Replace` except that it uses a regular expression.

The following replaces "cat" with "dog". Unlike with `string.Replace`, "catapult" won't change into "dogapult", because we match on word boundaries:

```
string find = @"\bcat\b";
string replace = "dog";
Console.WriteLine (Regex.Replace ("catapult the cat", find, replace));

OUTPUT: catapult the dog
```

The replacement string can reference the original match with the `$0` substitution construct. The following example wraps numbers within a string in angle brackets:

```
string text = "10 plus 20 makes 30";
Console.WriteLine (Regex.Replace (text, @"\d+", @"<$0>"));

OUTPUT: <10> plus <20> makes <30>
```

You can access any captured groups with `$1`, `$2`, `$3`, and so on, or `${`*`name`*`}` for a named group. To illustrate how this can be useful, consider the regular expression in the previous section that matched a simple XML element. By rearranging the groups, we can form a replacement expression that moves the element's content into an XML attribute:

```
string regFind =
  @"<(?'tag'\w+?).*>" +  // lazy-match first tag, and name it 'tag'
```

```
  @"(?'text'.*?)"     +  // lazy-match text content, name it 'text'
  @"</\k'tag'>";         // match last tag, denoted by 'tag'

string regReplace =
  @"<${tag}"          +  // <tag
  @"value=""""        +  // value="
  @"${text}"          +  // text
  @""""/>";              // "/>

Console.Write (Regex.Replace ("<msg>hello</msg>", regFind, regReplace));
```

Here's the result:

```
<msg value="hello"/>
```

MatchEvaluator Delegate

`Replace` has an overload that takes a `MatchEvaluator` delegate, which is invoked per match. This allows you to delegate the content of the replacement string to C# code when the regular expressions language isn't expressive enough:

```
Console.WriteLine (Regex.Replace ("5 is less than 10", @"\d+",
                   m => (int.Parse (m.Value) * 10).ToString()) );

OUTPUT: 50 is less than 100
```

In "Cookbook Regular Expressions", we show how to use a `MatchEvaluator` to escape Unicode characters appropriately for HTML.

Splitting Text

The static `Regex.Split` method is a more powerful version of the `string.Split` method, with a regular expression denoting the separator pattern. In this example, we split a string, where any digit counts as a separator:

```
foreach (string s in Regex.Split ("a5b7c", @"\d"))
  Console.Write (s + " ");     // a b c
```

The result, here, doesn't include the separators themselves. You can include the separators, however, by wrapping the expression in a *positive lookahead*. The following splits a camel-case string into separate words:

```
foreach (string s in Regex.Split ("oneTwoThree", @"(?=[A-Z])"))
  Console.Write (s + " ");    // one Two Three
```

Cookbook Regular Expressions

Recipes

Matching US Social Security number/phone number

```
string ssNum = @"\d{3}-\d{2}-\d{4}";

Console.WriteLine (Regex.IsMatch ("123-45-6789", ssNum));      // True

string phone = @"(?x)
  ( \d{3}[-\s] | \(\d{3}\)\s? )
    \d{3}[-\s]?
    \d{4}";

Console.WriteLine (Regex.IsMatch ("123-456-7890",   phone));   // True
Console.WriteLine (Regex.IsMatch ("(123) 456-7890", phone));   // True
```

Extracting “name = value” pairs (one per line)

Note that this starts with the *multiline* directive `(?m)`:

```
string r = @"(?m)^\s*(?'name'\w+)\s*=\s*(?'value'.*)\s*(?=\r?$)";

string text =
  @"id = 3
    secure = true
    timeout = 30";

foreach (Match m in Regex.Matches (text, r))
  Console.WriteLine (m.Groups["name"] + " is " + m.Groups["value"]);
id is 3 secure is true timeout is 30
```

Strong password validation

The following checks whether a password has at least six characters and whether it contains a digit, symbol, or punctuation mark:

```
string r = @"(?x)^(?=.* ( \d | \p{P} | \p{S} )).{6,}";
```

```
Console.WriteLine (Regex.IsMatch ("abc12", r));     // False
Console.WriteLine (Regex.IsMatch ("abcdef", r));    // False
Console.WriteLine (Regex.IsMatch ("ab88yz", r));    // True
```

Lines of at least 80 characters

```
string r = @"(?m)^.{80,}(?=\r?$)";

string fifty = new string ('x', 50);
string eighty = new string ('x', 80);

string text = eighty + "\r\n" + fifty + "\r\n" + eighty;

Console.WriteLine (Regex.Matches (text, r).Count);   // 2
```

Parsing dates/times (N/N/N H:M:S AM/PM)

This expression handles a variety of numeric date formats—and works whether the year comes first or last. The `(?x)` directive improves readability by allowing whitespace; the `(?i)` switches off case sensitivity (for the optional AM/PM designator). You can then access each component of the match through the `Groups` collection:

```
string r = @"(?x)(?i)
 (\d{1,4}) [./-]
 (\d{1,2}) [./-]
 (\d{1,4}) [\sT]
 (\d+):(\d+):(\d+) \s? (A\.?M\.?|P\.?M\.?)?";

string text = "01/02/2008 5:20:50 PM";

foreach (Group g in Regex.Match (text, r).Groups)
  Console.WriteLine (g.Value + " ");
01/02/2008 5:20:50 PM 01 02 2008 5 20 50 PM
```

(Of course, this doesn't verify that the date/time is correct.)

Matching Roman numerals

```
string r =
  @"(?i)\bm*"         +
  @"(d?c{0,3}|c[dm])" +
  @"(l?x{0,3}|x[lc])" +
  @"(v?i{0,3}|i[vx])" +
  @"\b";

Console.WriteLine (Regex.IsMatch ("MCMLXXXIV", r));   // True
```

Removing repeated words

Here, we capture a named group called dupe:

```
string r = @"(?'dupe'\w+)\W\k'dupe'";

string text = "In the the beginning...";
Console.WriteLine (Regex.Replace (text, r, "${dupe}"));

In the beginning
```

Word count

```
string r = @"\b(\w|[-'])+\b";

string text = "It's all mumbo-jumbo to me";
Console.WriteLine (Regex.Matches (text, r).Count);   // 5
```

Matching a GUID

```
string r =
  @"(?i)\b"           +
  @"[0-9a-fA-F]{8}\-" +
  @"[0-9a-fA-F]{4}\-" +
  @"[0-9a-fA-F]{4}\-" +
  @"[0-9a-fA-F]{4}\-" +
  @"[0-9a-fA-F]{12}"  +
  @"\b";

string text = "Its key is {3F2504E0-4F89-11D3-9A0C-0305E82C3301}.";
Console.WriteLine (Regex.Match (text, r).Index);                    // 12
```

Parsing an XML/HTML tag

Regex is useful for parsing HTML fragments—particularly when the document might be imperfectly formed:

```
string r =
  @"<(?'tag'\w+?).*>"  +  // lazy-match first tag, and name it 'tag'
  @"(?'text'.*?)"      +  // lazy-match text content, name it 'textd'
  @"</\k'tag'>";          // match last tag, denoted by 'tag'

string text = "<h1>hello</h1>";

Match m = Regex.Match (text, r);

Console.WriteLine (m.Groups ["tag"]);       // h1
Console.WriteLine (m.Groups ["text"]);      // hello
```

Splitting a camel-cased word

This requires a *positive lookahead* to include the uppercase separators:

```
string r = @"(?=[A-Z])";

foreach (string s in Regex.Split ("oneTwoThree", r))
  Console.Write (s + " ");    // one Two Three
```

Obtaining a legal filename

```
string input = "My \"good\" <recipes>.txt";

char[] invalidChars = System.IO.Path.GetInvalidFileNameChars();
string invalidString = Regex.Escape (new string (invalidChars));

string valid = Regex.Replace (input, "[" + invalidString + "]", "");
Console.WriteLine (valid);

My good recipes.txt
```

Escaping Unicode characters for HTML

```
string htmlFragment = "© 2007";

string result = Regex.Replace (htmlFragment, @"[\u0080-\uFFFF]",
                m => @"&#" + ((int)m.Value[0]).ToString() + ";");

Console.WriteLine (result);        // &#169; 2007
```

Unescaping characters in an HTTP query string

```
string sample = "C%23 rocks";

string result = Regex.Replace (
    sample,
    @"%[0-9a-f][0-9a-f]",
    m => ((char) Convert.ToByte (m.Value.Substring (1), 16)).ToString(),
    RegexOptions.IgnoreCase
);

Console.WriteLine (result);   // C# rocks
```

Parsing Google search terms from a web stats log

You should use this in conjunction with the previous example to unescape characters in the query string:

```
string sample =
  "http://google.com/search?hl=en&q=greedy+quantifiers+regex&btnG=Search";

Match m = Regex.Match (sample, @"(?<=google\..+search\?.*q=).+?(?=(&|$))");

string[] keywords = m.Value.Split (
  new[] { '+' }, StringSplitOptions.RemoveEmptyEntries);

foreach (string keyword in keywords)
  Console.Write (keyword + " ");       // greedy quantifiers regex
```

Regular Expressions Language Reference

Tables 25-2 through 25-12 summarize the regular expressions grammar and syntax supported in the .NET implementation.

Table 25-2. Character escapes

Escape code sequence	Meaning	Hexadecimal equivalent
`\a`	Bell	`\u0007`
`\b`	Backspace	`\u0008`
`\t`	Tab	`\u0009`
`\r`	Carriage return	`\u000A`
`\v`	Vertical tab	`\u000B`
`\f`	Form feed	`\u000C`
`\n`	Newline	`\u000D`
`\e`	Escape	`\u001B`
`\`*`nnn`*	ASCII character *nnn* as octal (e.g., `\n052`)	
`\x`*`nn`*	ASCII character *nn* as hex (e.g., `\x3F`)	
`\c`*`l`*	ASCII control character *l* (e.g., `\cG` for Ctrl-G)	
`\u`*`nnnn`*	Unicode character *nnnn* as hex (e.g., `\u07DE`)	
`\`*`symbol`*	A nonescaped symbol	

Special case: within a regular expression, `\b` means word boundary, except in a `[ ]` set, in which `\b` means the backspace character.

Table 25-3. Character sets

Expression	Meaning	Inverse ("not")
`[abcdef]`	Matches a single character in the list	`[^abcdef]`
`[a-f]`	Matches a single character in a *range*	`[^a-f]`
`\d`	Matches a decimal digit Same as `[0-9]`	`\D`
`\w`	Matches a *word* character (by default, varies according to `CultureInfo.CurrentCulture`; for example, in English, same as `[a-zA-Z_0-9]`)	`\W`
`\s`	Matches a whitespace character Same as `[\n\r\t\f\v ]`	`\S`
`\p{`*`category`*`}`	Matches a character in a specified *category* (see Table 25-4)	`\P`
`.`	(Default mode) Matches any character except `\n`	`\n`
`.`	(`SingleLine` mode) Matches any character	`\n`

Table 25-4. Character categories

Quantifier	Meaning
`\p{L}`	Letters
`\p{Lu}`	Uppercase letters
`\p{Ll}`	Lowercase letters
`\p{N}`	Numbers
`\p{P}`	Punctuation
`\p{M}`	Diacritic marks
`\p{S}`	Symbols
`\p{Z}`	Separators
`\p{C}`	Control characters

Table 25-5. Quantifiers

Quantifier	Meaning
`*`	Zero or more matches
`+`	One or more matches
`?`	Zero or one match
`{n}`	Exactly *n* matches
`{n,}`	At least *n* matches
`{n,m}`	Between *n* and *m* matches

The `?` suffix can be applied to any of the quantifiers to make them *lazy* rather than *greedy*.

Table 25-6. Substitutions

Expression	Meaning
`$0`	Substitutes the matched text
`$group-number`	Substitutes an indexed *group-number* within the matched text
`${group-name}`	Substitutes a text *group-name* within the matched text

Substitutions are specified only within a replacement pattern.

Table 25-7. Zero-width assertions

Expression	Meaning
`^`	Start of string (or line in *multiline* mode)
`$`	End of string (or line in *multiline* mode)
`\A`	Start of string (ignores *multiline* mode)
`\z`	End of string (ignores *multiline* mode)
`\Z`	End of line or string
`\G`	Where search started
`\b`	On a word boundary
`\B`	Not on a word boundary
`(?=`*`expr`*`)`	Continue matching only if expression *`expr`* matches on right (*positive lookahead*)
`(?!`*`expr`*`)`	Continue matching only if expression *`expr`* doesn't match on right (*negative lookahead*)
`(?<=`*`expr`*`)`	Continue matching only if expression *`expr`* matches on left (*positive lookbehind*)
`(?<!`*`expr`*`)`	Continue matching only if expression *`expr`* doesn't match on left (*negative lookbehind*)
`(?>`*`expr`*`)`	Subexpression *`expr`* is matched once and not backtracked

Table 25-8. Grouping constructs

Syntax	Meaning
`(`*`expr`*`)`	Capture matched expression *`expr`* into indexed group
`(?`*`number`*`)`	Capture matched substring into a specified group *`number`*
`(?'`*`name`*`')`	Capture matched substring into group *`name`*
`(?'`*`name1-name2`*`')`	Undefine *`name2`* and store interval and current group into *`name1`*; if *`name2`* is undefined, matching backtracks
`(?:`*`expr`*`)`	Noncapturing group

Table 25-9. Back references

Parameter syntax	Meaning
`\`*`index`*	Reference a previously captured group by *`index`*
`\k<`*`name`*`>`	Reference a previously captured group by *`name`*

Table 25-10. Alternation

Expression syntax	Meaning
`|`	Logical *or*
`(?(`*expr*`)`*yes*`|`*no*`)`	Matches *yes* if expression matches; otherwise, matches *no* (*no* is optional)
`(?(`*name*`)`*yes*`|`*no*`)`	Matches *yes* if named group has a match; otherwise, matches *no* (*no* is optional)

Table 25-11. Miscellaneous constructs

Expression syntax	Meaning
`(?#`*comment*`)`	Inline comment
`#`*comment*	Comment to end of line (works only in `IgnorePatternWhitespace` mode)

Table 25-12. Regular expression options

Option	Meaning
`(?i)`	Case-insensitive match (“ignore” case)
`(?m)`	Multiline mode; changes `^` and `$` so that they match beginning and end of any line
`(?n)`	Captures only explicitly named or numbered groups
`(?c)`	Compiles to Intermediate Language
`(?s)`	Single-line mode; changes meaning of “.” so that it matches every character
`(?x)`	Eliminates unescaped whitespace from the pattern
`(?r)`	Searches from right to left; can’t be specified midstream