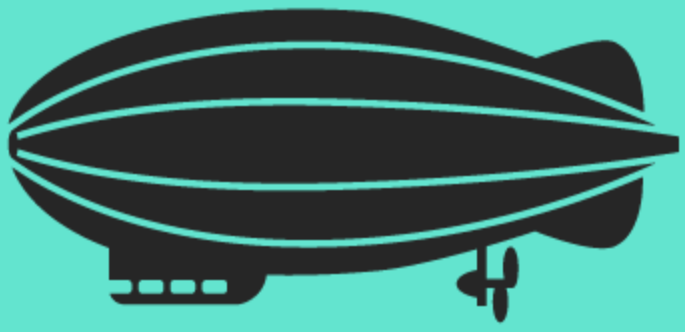


Regular Expressions Succinctly

by Joe Booth

Syncfusion | Technology Resource Portal

Regular Expressions Succinctly

By
Joe Booth

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: Thomas Garrett
Copy Editor: Benjamin Ball
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Morgan Cartier Weston, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Joseph D. Booth has been programming since 1981 in a variety of languages, including BASIC, Clipper, FoxPro, Delphi, Classic ASP, Visual Basic, Visual C#, and the .NET framework. He has also worked in various database platforms, including DBASE, Paradox, Oracle, and SQL Server.

He is the author of six computer books on Clipper and FoxPro programming, Network Programming, and client/server development with Delphi. He also wrote several third-party developer tools, including CLIPWKS, which allowed the ability to programmatically create and read native Lotus and Excel spreadsheet files from Clipper applications. He is the author of *Visual Studio Add-ins Succinctly* from Syncfusion.

Joe has worked for a number of companies including Sperry Univac, MCI-WorldCom, Ronin, Harris Interactive, Thomas Jefferson University, People Metrics, and Investor Force. He is one of the primary authors of Results for Research (market research software), PEPSys (industrial distribution software), and a key contributor to AccuBuild (accounting software for the construction industry).

He also has a background in accounting, having worked as a controller for several years in the industrial distribution field, but his real passion is computer programming.

In his spare time, Joe is an avid tennis player and a crazy soccer player (he plays goalie). He also practices yoga and martial arts, and holds a brown belt in Judo. You can visit his website at www.joebooth-consulting.com.

Preface

Target Audience

This book is for developers who are currently using Microsoft Visual Studio and want to learn how to use regular expressions in their C# and VB.NET code. It assumes you are comfortable programming in C# and have at least seen regular expressions, and have a notion of what they are used for. The book is intended to help you understand them and start using them more confidently to solve programming problems.

Tools Needed

In order to be able to follow along with all of the examples in this book, you will need Microsoft Visual Studio 2010 or later. We will create an application for testing regular expressions with Microsoft tools.

You can also use online testing tools (such as regex101.com or regexhero.net), so you'll need a browser and internet access if you want to try out regular expressions in languages other than the .NET framework. Although this book is primarily targeted on the Microsoft platform, almost all of the expressions will work in other implementations.

Formatting

Throughout the book, I have used the following formatting conventions:

Note: Notes or ideas about the current topic, and problems you might encounter.

Tip: Ideas, tips, and suggestions.

Code Blocks

Any code samples in the book will be formatted as shown below:

```
string pattern = @"[0-9A-F]";       //       Pattern for hex number
Regex theRegex = new Regex(pattern);
```

Introduction

What exactly are regular expressions? Those cryptic strings you'll see in code and wonder, "What on earth is that?" What does **`\b([1-9]|1[0-2]|0[1-9]){1}(:[0-5][0-9][aApP][mM]){1}\b`** mean? In this book, we hope to provide enough guidance to not only interpret the above (and other expressions), but also to write and use your own.

Simply put, a regular expression (abbreviated to regex hereafter), is a pattern of characters that are used to find matches in a larger string or text. The pattern describes what a string that matches should look like. For example, if we have the following text, "the gray hound chased the red fox for over a full mile," we might want to find the color gray in the text. A regex to find the word "gray" in the text would simply be the characters **`'gray'.`** Of course, almost every programming language can do that.

Microsoft SQL

```
select charindex('gray','the gray hound chased the red fox for over a full mile') as foundAt
```

.NET (C#)

```
string theText = "the gray hound chased the red fox for over a full mile";
int x = theText.IndexOf("gray");
```

However, this assumes you know exactly the text you are looking for. Suppose this text came from friends in England, and read the following instead:

"The grey hound chased the red fox for over a full kilometer."

We can get a bit more complicated to make sure we handle the variations gray and grey.

```
string theText = "the gray hound chased the red fox for over a full mile";
int x = theText.IndexOf("gray");
if (x<0)
{
    x =  theText.IndexOf("grey");

}
```

```
declare @theText varchar(200)
set @theText = 'the grey hound chased the red fox for over a full kilometer'

select
case when charindex('gray',@theText) >0
     then charindex('gray',@theText)
     else charindex('grey',@theText)

     end as FoundAt
```

The regex string to accomplish the same function is simply **‘gr[ae]y’**. This says, find the letters **gr**, followed by either an **a** or an **e**, and finally a **y**. Maybe there is something to regex after all?

Chapter 1 Text Processing

Computers are very good at dealing with structured data, but cannot come anywhere close (yet) to how good people are at dealing with text. For example, if the following text were recorded in a help desk system:

```
Hi,

I called Jon on Tuesday, March 25th at 7pm and expressed a concern about my
slow times accessing www.cnn.com. He said he would fix it, but I never heard
back. Can someone contact me at Kellie.Booth@if.com ASAP? What does Ctrl-F5
mean, by the way?

Thanks

Kellie
```

Without much effort, a person can read that text and easily find the person Kellie talked to, the problem website, the date, and the email address. However, such a simple task for a person is rather difficult for a computer. What happens if the date was written as Mar-25 instead? Or if the website didn't include the "www" prefix? Or if it included "http:" in the address?

Patterns

How do people do it so well? Essentially, we understand patterns and can easily handle missing information. We know a date may or may not have the day as part of it, and a month name might be spelled out or might be abbreviated. We know that a website typically is a word or more (without spaces), followed by a period and a common abbreviation like .com, .net, or .org.

Constructing a regex pattern is a matter of understanding the rules of what some text looks like and then describing those rules in the regex pattern language. Throughout the rest of this book, we are going to work with the above example, a help system that contains a date when a message was received and the text contents of the message. We will explore using regex patterns to parse these text messages and extract dates, times, URLs, email addresses, etc.

Here is another example message from our help system:

```
Hey jerks,

I called on Mar 15, spoke to Suzie, and complained about not being able to
access livedates.net after 7:30PM. What the heck is up? Get back to me,
fred@mycollege.edu. Or call me at 619-555-1212. Hopefully, somebody will read
this…

Very upset, Fred!
```

Our ongoing task throughout this book will be to use regular expressions to figure out how to read these messages, and determine when the call was made, who the call was made to, which website they are having issues with, and their email address or phone number.

Building Patterns

As we work through the book, we will see a variety of ways that a regex pattern can match a result. The best way to build your pattern is to step away from the computer and write out in the steps in English of how you recognize the pattern. Once you describe your search rules in English, you can then convert those rules to the appropriate regex patterns that match the logic you would employ to search for the text yourself.

For example, let's say we are looking for price information in an online ad. We want to find a car for sale, but we don't want to spend more than $10,000. Here are the rules we might employ in English to find the price.

Table 1: Pattern Rules

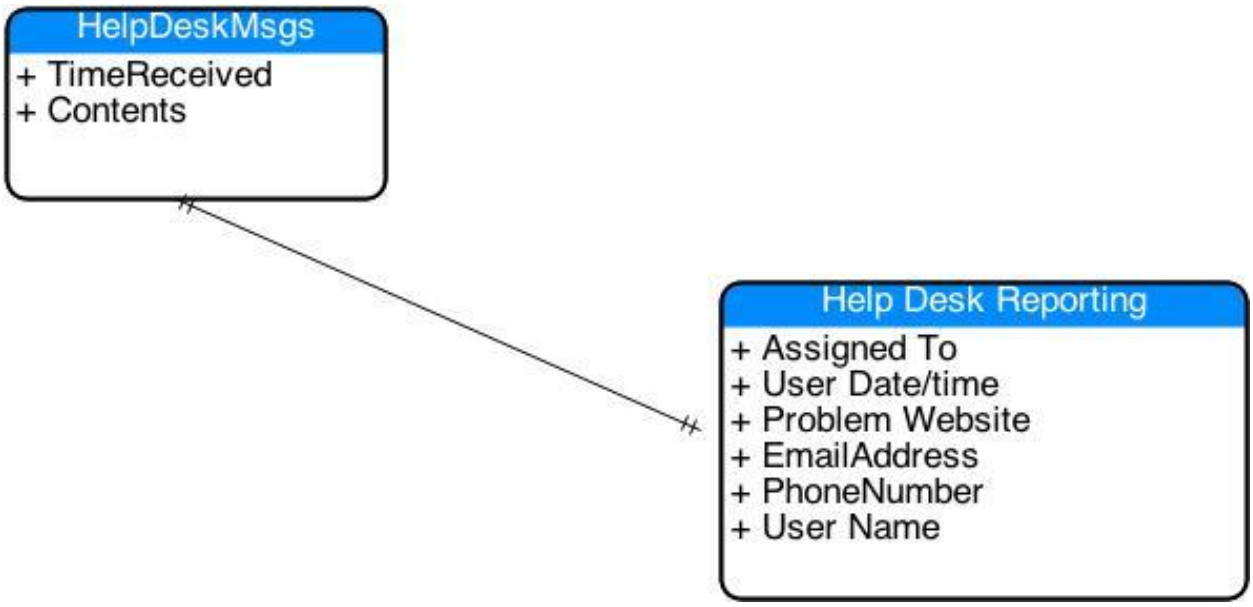
English Rule	Regex Pattern
Match all text up until the first occurrence of the $ symbol.	`.*\$`
Now we need some numbers.	`\d+`
There may be a comma.	`,?`
And some more numbers.	`d+`

While you employ these steps without much thought, you'll need to figure them out in order to convert them to the regex pattern language to teach the computer what you do intuitively.

Note: Don't worry about the regex pattern side yet; those characters will make much more sense in a few chapters.

Our Help Desk

For the sake of our example, we'll assume our help desk is staffed by three people named John, Susan, and Bill. SMS text messages or emails sent to the help system are automatically logged to a database table holding only two fields: the date and time the message was received, and the text of the message. The company wants to start reporting a summary of these text messages, to whom the message was sent and when, the recipient's contact information, and other relevant information.



Help Desk Workflow

A person can easily read the messages from the help desk messages table and extract the information into the help desk reporting table. As we work through this book, our aim is to be able to extract most of the information from the help desk messages table to create an entry in the reporting table.

Chapter 2 Using Regular Expressions in .NET

The concept of regular expressions first arose in the 1950s when American mathematician Stephen Kleene designed a simple algebraic system to understand how the brain can produce complex patterns. In 1968, this pattern algebra was described in the *Communications of the ACM* and implemented in several UNIX tools, particularly GREP, a UNIX tool for searching for files. When the Perl programming language from the late 1980s integrated regular expressions into the language, adding features to the original algebra, it brought regex parsing into the mainstream programming community.

Regex processing exists in many languages, so the patterns and rules should work fairly consistently no matter what tool you are using. There are some subtle differences we will point out as we explore examples, but most implementations support the language elements the same way.

Microsoft Regex

For purposes of this book, we are going to use the regex processing engine provided by Microsoft in the .NET framework. We will use C# as our base language, however, with minor differences; the code examples will work in VB.NET. The patterns should work in .NET, PHP, Python, etc.

The Microsoft .NET Framework since version 1.1 has supported regular expressions in the ***System.Text.RegularExpressions*** assembly. This assembly needs to be included in any application you write in order to take advantage of regular expressions.

Regex Class

The regex class can be used to create an object that will allow you set a pattern and some options, and then apply this pattern to strings. You can search for strings, split a string to an array, or even replace strings within a text.

You can use the regex class in two ways. The first is by creating a regex object. You do this by passing a regex pattern and optionally some regex settings. However, one drawback to using this approach is that the regex pattern cannot be changed once the object has been instantiated.

You can also use the regex class as a static class (or shared in Visual Basic). When using this approach, you will need to pass both the pattern and the search string to the regex call.

Which approach to use depends on how you are using the regex. If you are reading a text file line by line, and using the same regex to split each line into words or columns, then it would be beneficial to create a regex object and let the framework compile and cache the expression. You'll get better performance from the regex engine using this approach.

If you take a text string and apply several different regex patterns to it (such as our examples where we are searching through help messages), then using the static regex would make more sense, since each regex is used once and you don't gain much from keeping a compiled version around.

Note: The Microsoft engine will compile regular expressions and keep them in the cache (which defaults to 15 items). If you reuse a pattern, it could already be in the cache, which helps performance. However, every time you create a regex object (rather than the static class), that regex will be recompiled and added to the cache.

Constructor

To create a regex object, you need to pass the pattern to the regex at a minimum. The basic syntax is:

```
string pattern = @"[0-9A-F]";        //       Pattern for hex digit
Regex theRegex = new Regex(pattern);
```

Tip: Regex strings often contain characters that might normally be escaped in a string literal, particularly the \ character. By preceding your regular expressions string with the symbol @, you can avoid the need to escape any characters.

You can also pass an enumerated list of regex options (separate each option with the | character). To take the above constructor code and add options to make it case-insensitive, as well as process the string from right to left, we could combine the following options (we will discuss the options in more detail in a later chapter):

```
string pattern = @"[0-9A-F]";        //       Pattern for hex digit
Regex theRegex = new Regex(pattern,RegexOptions.IgnoreCase |
RegexOptions.RightToLeft);
```

Note: .NET 4.5 has added a third optional parameter, a TimeSpan option. By default, a regex will not time out, but rather keep searching. If you need the regex to only run for a defined period of time, you can add the TimeSpan option to the regex when the object is constructed.

Regex Options

RegexOptions is an enumerated type of the various options you can specify to control how the regex is processed. Some common options are described in Table 2:

Table 2: Common Regex Options

Option	Description
None	No options specified, default regex processing.
IgnoreCase	Regex patterns are normally case-sensitive; this option makes the regex case-insensitive.
Multiline	Certain metacharacters ($ and ^) mean the beginning and end of a string; setting this option changes their meanings to the beginning and end of a line.
Singleline	The dot metacharacter does not match the linefeed; using this option causes the linefeed to be matched by the dot as well.

There are additional options which we will cover in later chapters of the book.

Regex Methods

There are both regular class methods and static methods, depending on whether you are using a created object or a static object reference. The methods are similar, the main difference being that when using the static method, you need to pass the regex pattern as a parameter.

IsMatch()

The IsMatch() method returns a Boolean value indicating whether or not the text matches the regular expression. Most often, it is used to validate that a particular string looks like the pattern, such as a credit card number, phone number, etc.

IsMatch(string Searchtext [, int position])

When used against a regular object, there are two parameters. The first is the string to search, and the second is the optional parameter indicating a position in the string at which to begin searching.

IsMatch(string Searchtext, string pattern [, RegexOptions] [, TimeSpan])

When the static method is called, the search text and regex pattern are required. The regex options (enumerated list separated by the | pipe) and a TimeSpan object are optional (.NET 4.5 onward).

```
      string pattern = @"[0-9A-F][0-9A-F]";      //  Pattern for hex digits
      Regex theRegex = new Regex(pattern);
      if (theRegex.IsMatch("B4") )
          {  // Found a hex number
          };
```

Typically **IsMatch()** is used when you need no further manipulation of the string or simply want to validate that the string looks like the type of data you are expecting.

Match()

The Match() method works very similarly to IsMatch(), except that it returns a match object rather than a Boolean value. A match object is defined by a class within the regex assembly that provides details about the match found in the string.

Match(string Searchtext [, int position],[int NumberOfChars])

When used against a regular object, there are three parameters. The first is the string to search. The second is an optional parameter for the position within the string at which to start searching. The third optional parameter is the number of characters to search within the string.

Match(string Searchtext, string pattern [, RegexOptions] [, TimeSpan])

When the static method is called, the search text and regex pattern are required. The regex options (enumerated list separated by the | pipe) and (.NET 4.5 onward), a TimeSpan object are optional.

The returned Match object contains information about the match. The key properties are:

- Success: Boolean value indicating whether the search was successful or not.
- Index: Where in the search text the match was found.
- Length: Size of the match text.
- Value: The content of the matched text.

```
 string pattern = @"(\([2-9]|[2-9])(\d{2}|\d{2}\))(-|.|\s)?\d{3}(-
|.|\s)?\d{4}";
 string source = "Please call me at 610-555-1212 ASAP !";
 string cPhone = "";


 Match theMatch = Regex.Match(source, pattern);
  if (theMatch.Success)
     {
       int endindex = theMatch.Length;
       cPhone = source.Substring(theMatch.Index, endindex);
     }
```

In this example, we are searching for a phone number pattern, then using the match object's properties (i.e. Index and Length) to extract the phone number string from the larger source string. There are additional properties and methods to the Match object which we will cover in later chapters.

Matches()

The Matches() method is very similar to the Match() method, except it returns a collection of match objects (or an empty collection is no matches found).

Matches(string Searchtext [, int position])

When used against a regular object, there are two parameters. The first is the string to search and the second optional parameter is a position within the string to start searching at.

Matches(string Searchtext, string pattern [, RegexOptions] [, TimeSpan])

When the static method is called, the search text and regex pattern are required. The regex options (enumerated list separated by the | pipe) and a TimeSpan object (.NET 4.5 onward) are optional.

```
string pattern = @"(\([2-9]|[2-9])(\d{2}|\d{2}\))(-|.|\s)?\d{3}(-
|.|\s)?\d{4}";
string source = "Please call me at home 610-555-1212 or my cell 610-867-5309
ASAP !";
string cPhone = "";
var phones = new List<string>();


foreach (Match match in Regex.Matches(source, pattern))
       {
           int endindex = match.Length;
           cPhone = source.Substring(match.Index, endindex);
           phones.Add(phones);
       }
```

Split()

The Split() method in the regex assembly is very similar to the Split() method in the System.string class, except that a regex pattern is used to split the string into an array of strings.

Split(string Searchtext [, int position],[int NumberOfChars])

When used against a regular object, there are three parameters. The first is the string to split into an array of strings, and the second is the optional parameter indicating a position in the string at which to begin searching. The third optional parameter is the number of characters to search within the string.

Split(string Searchtext, string pattern [, RegexOptions] [, TimeSpan])

When the static method is called, the search text and regex pattern are required. The regex options (enumerated list separated by the | pipe) and a TimeSpan object (.NET 4.5 onwards) are optional.

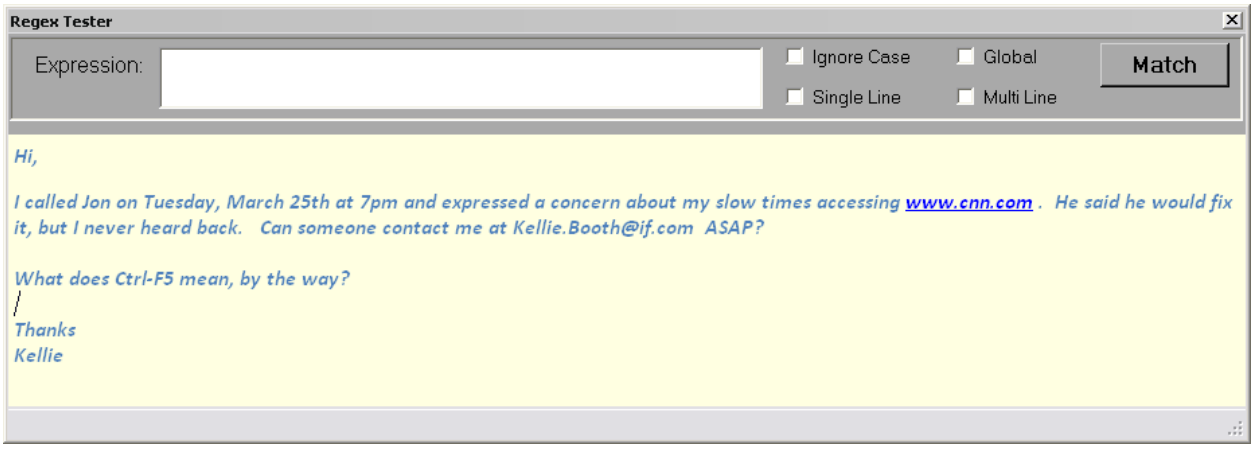
For example, you might use code similar to below to split a large text into individual sentences, and then each sentence into words.

```
string source = "When I try this website, the browser locks up";
Regex WordSplit = new Regex(@"[^\p{L}]*\p{Z}[^\p{L}]*";);

string thePattern = @"(?<=[\.!\?])\s+";
string[] sentences = Regex.Split(source,thePattern);

foreach (string sentence in sentences)
   {
      string[] Words = WordSplit.Split(sentence);
   }
```

There are a few additional methods and properties in the regex assembly which will be introduced in later chapters.

Regex Tester Program

In order to test out some of the regex patterns in this book, we are going to create a simple Windows Forms application for testing expressions. There are also websites that provide similar functionality, such as regex101.com, which provides a regex tester for JavaScript, PHP, or Python. While the patterns described within this book will work with each of the various regex engines, you should use this regex application if you want to test with the .NET Regular Expression framework, or try the website if you work with a different programming environment.



Regex Tester

Layout

The solution is available as part of this book; you can also create it yourself.

1. Create a Windows Forms application.
2. Add a **Split Container**.
3. Set the Orientation to **Horizontal**.
4. In the top box, add a **Panel**, and in the bottom box, add a **Rich Text Edit** box.

Layout

Panel

The panel at the top of the form holds the regular expression text and option boxes. You can name the controls anything, but the following references the control names in Table 3.

Table 3: Panel Control Elements

Element	Property	Value
Textbox	Name	TB
	Font	Courier New, 9.75pt
Checkbox	Name	ICCB
	Text	Ignore Case
Checkbox	Name	GBCB
	Text	Global
Checkbox	Name	SLCB

Element	Property	Value
	Text	Single Line
Checkbox	Name	MLCB
	Text	Multiline
Status bar	Name	SS
	Items	Add a Status label called SLAB

Rich Edit Text Box

The Rich Edit Text Box will be used to display the input string and the search text result. The code finds all matching patterns and highlights them within the RTF edit box. Set the property name to **rtb**. Also set the anchors properties to top, bottom, left, and right to have the text box fill the entire lower panel.

Code

Once the form is created, add the following code (feel free to change the colors if you'd like).

```
public partial class MainForm : Form
{

    Color BGColor = SystemColors.Info;
     Color FGColor = Color.Navy;
     Color BGHighlight = Color.Turquoise;
     Color FGHighlight = Color.Black;

     public MainForm()
     {
         InitializeComponent();
         rtb.BackColor = BGColor;
         rtb.ForeColor = FGColor;
     }
```

Double-click the button to generate the OnClick event, and add the following code to the form to handle the event:

```
     private void BN_Click(object sender, EventArgs e)
     {
         SLAB.Text = "";
         // Reset all colors.
         ResetRichTextBox();
```

```
            string pattern = TB.Text.Trim();
            string source = rtb.Text;

            RegexOptions theOpts = RegexOptions.None;

            if (MLCB.Checked) { theOpts = theOpts | RegexOptions.Multiline; }
            if (SLCB.Checked) { theOpts = theOpts | RegexOptions.Singleline;
}
            if (ICCB.Checked) { theOpts = theOpts | RegexOptions.IgnoreCase;
}

            // If global, then iterate the Matches collection.
            if (GBCB.Checked)
            {
                try
                {
                    foreach (Match match in Regex.Matches(source, pattern,
theOpts))
                    {
                        HighLightResult(match);
                    }
                }
                catch (Exception ex)
                {
                     SLAB.Text = ex.Message;
                }
            }
            else
            {
                try
                {
                    Match theMatch = Regex.Match(source, pattern, theOpts);
                    if (theMatch.Success)
                    {
                        HighLightResult(theMatch);
                    }
                    else
                    {
                        SLAB.Text = "Not found...";
                    }
                }
                catch (Exception ex)
                    {
                        SLAB.Text = ex.Message;
                    }

            }
        }
```

Then add a couple of internal functions:

```
        private void HighLightResult(Match OneMatch)
        {
            int endindex = OneMatch.Length;
            rtb.Select(OneMatch.Index, endindex);
            rtb.SelectionBackColor = BGHighlight;
            rtb.SelectionColor = FGHighlight;
        }

        private void ResetRichTextBox()
        {
            rtb.SelectAll();
            rtb.SelectionBackColor = BGColor;
            rtb.SelectionColor = FGColor;
        }

        private void TB_TextChanged(object sender, EventArgs e)
        {
            ResetRichTextBox();
        }
```

Feel free to customize the code elements and such, but be sure to either create the application or download it; it will be very handy as you explore the regular expressions described throughout the book.

Chapter 3 Character Sets

A regex pattern is a string expression that describes how to find matching text. The simplest pattern is simply a letter or number we want to match. For example, in our help system text, we could find the time by using the regex pattern **`7pm.`** While searching specifically for 7pm is not very useful, it illustrates the first rule of a regex.

Literal Characters

Regex has a number of characters that have special meaning called metacharacters—we will explore how to use them later in this chapter. The metacharacters are:

```
^ $ . | {  } [ ] ( ) * + ? \
```

All other characters in the pattern are literals. The regex search engine expects to find the exact literal character somewhere in the string. In our example above, **`7pm`** contains three literal characters, and will only match the exact text 7pm in the search string. Note that, by default, regex is case-sensitive, so 7PM would not match. Although we can add an option to have the regex engine ignore case, let's look at a slightly different approach to solving the problem.

Square Brackets []

We are going to introduce our first metacharacters, the square brackets []. The square brackets are used to provide a list of potential matching characters at a position in the search text. So, to search for 7pm, case-insensitive, we could use the regex pattern **`7[Pp][Mm]`**. This pattern tells the regex to find the number 7, followed by either an upper case or lower case P, followed by an upper or lower case M. Using this pattern, 7pm, 7Pm, 7pM, or 7PM would all be found. However, the [] will only match one character in the list, so 7PPM would not match the pattern.

We can also adjust our pattern to find any time between 1 and 9 a.m. or 1 and 9 p.m. (we will cover how to get 10, 11, or 12 in a later chapter). We are now going to tweak our regex pattern to find a number between 1 and 9, followed by an upper or lower case A or P, followed by an upper or lower case letter M. The resulting pattern is **`[123456789][aApP][Mm]`**. This pattern will handle any time beginning with a 1 through 9, followed by AM or PM (regardless of case).

Ranges

In addition to listing out all the possible characters within the brackets, you can include a dash character to indicate a range of characters. For example, we can change our time search pattern from **`[123456789][aApP][Mm]`** to **`[1-9][aApP][Mm].`** This shorthand notation makes the regex a bit easier to write. You can also combine ranges between the brackets. For example, the following regex will find a two digit hex number **`[0-9A-F][0-9A-F]`**. We could add **`a-f`** if we wanted to include lower case letters as well.

You can also combine ranges and characters together, so the regex **`[0-9ABCDEF]`** is the same as **`[0-9A-F]`** to return a single hex digit.

Note that if you want to include the dash within the brackets (as a literal character), then it must come at the beginning or end of the string. To test for a +, -, or digit, you could use the following **`[-+0-9]`** pattern, which says accept a dash (literal character) or a plus sign, or any number between 0 and 9.

Non-printable characters

There are certain characters than cannot be entered into a regex pattern, such as a tab, a new line character. These characters are known as character escapes and allow just about any character to be searched for within a text.

- \a - Matches the bell character.
- \b - Matches a backspace (when used between brackets).
- \t - Matches a tab character.
- \r - Matches a carriage return.
- \n - Matches a new line; most Windows files use \r\n to indicate the end of a line.
- \f - Matches Form feed.
- \v - Matches a vertical tab.
- \xNN - Matches the ASCII character represented by the hex value NN.
- \cX - Matches an ASCII control character, such as \cC is Ctrl-C.

When the \ is followed by another character that is a not an escape character, it matches that character, even if the character normally has special meaning in the regex. For example, as we will see later, the asterisk has a special value inside a regex pattern. However, the * character overrides the special value (within brackets) and matches the asterisk character.

Some other characters do not have ASCII equivalents; however, you can use the Unicode value to search for those characters. For example, if we were looking for the Euro symbol €, we could use the regex pattern **`\u20AC.`** Not all regex implementations support the \u metacharacter, although JavaScript and the .NET framework do support it. You can use the following Unicode symbols to find various currency codes in a text:

- Euro € - **`\u20AC`**

- British pound £ - **\u00A3**
- Yen ¥ **-\u00A5**
- Dollar sign $ - **\$** or **\u0024** or **\x24**

To find any of the above currency symbols, the **[\u20AC\u00A3\u00A5\$]** should do the trick. Note that the $ is a metacharacter in regex, which is why we escape it with the backslash character.

EMCAScript is a scripting language specification that web client languages, such as JavaScript, are based on. One notable difference is that EMCAScript does not support Unicode in its regex processing rules.

Negation

When you build a character set within the brackets, it is sometimes easier to specify what values you want to exclude. This is what the ^ metacharacter within the square bracket allows you to do. The pattern **[aeiou]** would find all vowels, and the pattern **[^aeiou]** would find all consonants (i.e. anything that is not a vowel).

The negation character applies to everything within the square brackets. For example, if you negate the **0-9A-F** pattern, i.e. **[^0-9A-F]**, you will find everything that is not a valid hex number.

Shortcuts

In addition to ranges and lists of characters, there are also a number of shortcut metacharacters available for some common character patterns.

1.1.1.1.1 Digits

The **\d** metacharacter means to find any digits, it is the same as **[0-9]**. You can negate that by using the uppercase **\D** to find any non-digits, the same as **[^0-9]**.

1.1.1.1.2 Alphanumeric (word characters)

The **\w** metacharacter means to match any word character (any uppercase or lowercase letter, any digit, and the underscore). It is the same as **[a-zA-Z0-9_].**The **\W** metacharacter negates that, and matches any non-word character (i.e. **[^a-zA-Z0-9_]**).

1.1.1.1.3 White space

The **\s** metacharacter means to match any white space character in a string. White space includes carriage return, line feed, tab, form feed, and the space character, i.e. **[\r\n\t\f].** The **\S** metacharacter is the negation, and matches any non-white space character.

The shortcut metacharacters can be used outside of the square brackets, so **\d** will find a digit, while **0-9** (without brackets) will look for the exact pattern 0-9.

The dot

The dot metacharacter matches any single character (with the exception of \n-linefeed). When regex was first designed, it was targeted toward lines from text files, which is why the linefeed character is not matched. In .NET, the single line regex option causes the dot to also match the linefeed (\n) character.

To actually match a dot or period, you need to escape it with the backslash. For example, matching a decimal number with two decimal places might look like this **`\d\d\.\d\d`** if you didn't escape the dot, it would match, but so would a 5 digit number without a decimal point, two digits, a dash, two more digits, etc.

Some examples

The character class regex patterns can be handy when you want to match a string of the same length as the pattern. In the next chapter, we will see how to match varying lengths. Let's review a couple examples.

Social Security Number

A social security number is a nine-digit number, grouped into three digits, a dash, two digits, another dash, and then four final digits. To express this pattern as a regular expression, we could use the following: **`[0-9][0-9][0-9][-][0-9][0-9][-][0-9][0-9][0-9][0-9].`**

ZIP+4 code

A ZIP+4 code is a postal code used by the United States to provide a very specific mailing address location. The pattern consists of five digits followed by a dash or space and followed by four more digits. **`\d\d\d\d\d[-\s][0-9][0-9][0-9][0-9]`**

Euro price

This example shows how to find a price expressed in euros in a text string.

`[\u20AC]\d\d\.[0-9][0-9]`

This pattern looks for the euro symbol, followed by two digits, a decimal point, and two more digits.

Summary

In this chapter, we covered literal characters and metacharacters. Literal characters match exactly, while metacharacters have special meaning. The metacharacters we covered are described in Table 4.

Table 4: Meta-characters

Character	Description
[]	Provide a list of possible characters to match
\a	Bell character
\b	Backspace (when between brackets)
\t	Tab character
\r	Carriage return
\n	New line
\f	Form feed
\v	Vertical tab
\xNN	ASCII character NN
\cX	Control character
\uXXX	Unicode character
^	Negates characters within brackets
\d	Any single digit
\D	Any non-digit
\w	Alphanumeric (word character)
\W	Not a word character
\s	White space
\S	Non white space
.	Matches anything (except linefeed)

Chapter 4 Quantifiers

In the previous chapter, we looked at the various options for matching characters within a search text; however, the examples assumed that we knew the exact number of characters we are searching for. In this chapter, we will explore how we can search for patterns of varying length.

Curly brackets {}

The curly brackets metacharacters allow you to specify the number of times a particular character set occurs. For a simple example, let's take our euro regex from the previous chapter and make it a more general purpose currency search.

First, we want a symbol, which in this case is the euro, British pound, or dollar sign. **`[\u20AC\u00A3$]`** followed by at least one digit, but no more than five digits, followed by a period, and two digits. **`\d{1,5}\.\d\d`**

The **`{1,5}`** following the **`\d`** character indicates we need at least one, but no more than five digits after the currency symbol and before the period. Our options with the curly brackets can be:

- {n} - Preceding character must occur exactly "n" times.
- {n,} - At least "n" times, but no upper limit.
- {n,m} - Between "n" and "m" times.

Note that using **`{,n}`** will not search for a value between 0 and "n" times, but rather be interpreted as a literal value. If you can accept 0 occurrences, you need to specify it exactly, i.e. **`{0,}.`**

Examples

To revisit some of the examples from the previous chapter, here are some examples.

Social Security number

Our SSN checker can be simplified from **`[0-9][0-9][0-9][-\.][0-9][0-9][-\.][0-9][0-9][0-9][0-9]`** to **`\d{3}-\d{2}-\d{4}.`**

Zip code

Our Postal ZIP+4 code checker can be improved from **`\d\d\d\d\d[-][0-9][0-9][0-9][0-9]`** to **`\d{5}-{0,1}\d{0,4}`**. However, even though this expression is improved, and will handle both five digit zip codes and ZIP+4, it is not perfect because it would match five digits followed by a dash, five digits followed by and dash, and then less than four digits.

Tip: Whenever you build a regex pattern, it is always advisable to run it in a test program to ensure it matches what you'd expect. As patterns get more and more complex, it is likely you'll find some unexpected matches.

Now combining our first two chapters, let's put together a regex pattern to see if we can extract the time from our messages. We know that a time starts with a number between 1 and 12, can optionally be followed by a colon, and then two digits, the first between 0 and 5 and the second between 0 and 9. Finally, it should end with AM or PM, case-insensitive.

Table 5: Time Pattern

English rule	Regex pattern
Might start with 0 or 1	`[0-1]{0,1}`
Need a number	`\d`
And a colon	`:`
A number between 0 and 5	`[0-5]`
A number between 0 and 9	`[0-9]`
An optional whitespace character	`\s{0,1}`
Case-insensitive a or p	`[aApP]`
Case-insensitive m	`[Mm]`

The resulting regex pattern is **`[0-1]{0,1}\d:[0-5][0-9]\s{0,1}[aApP][Mm]`** and will match the following times: 7:30 p.m., 12:45 p.m., 6:45 a.m., etc. While it is not perfect, is it pretty close (and will get closer in later chapters). Unfortunately, it will also match 19:00 p.m., which is an invalid time, but still looks alright according to the defined pattern rules.

Greedy vs. Lazy Quantifiers

In addition to using the curly brackets to specify the number of occurrences you'll accept, there are other quantifiers available. However, to use these quantifiers, it is important to distinguish between greedy and lazy quantifiers. For example, let's assume we are parsing an HTML file and we want to remove the <> tags from the strings. The first line we encounter in our text is **Financial Report.** We need to write a regular expression pattern to find (and eventually remove) these tags.

Greedy

Our first attempt might look like this **<.{0,}>**, which says find a literal **<** sign, followed by 0 or more of any character, followed by a literal **>** sign. However, when processing that regex, we get the following result: Financial Report. In this case, the entire string was returned. This is an example of a greedy quantifier. A greedy quantifier takes the entire string and sees if it can match the pattern. Since the entire string matches the pattern, the regex returns the entire string (it begins with a **<** sign, contains any number of characters, and ends with a **>** sign).

By default, all quantifiers in curly brackets are greedy. They take the entire string and see if it matches. If not, they back track one character at a time until they find a match.

Greedy quantifiers are quite resource intensive because they test the entire string. If it does not match, the process backs up a character and checks to see if a match is found. The process also keeps track of where it is, in case it needs to go back to a prior location to pick up the search.

If our example was missing the final **>** character, the regex engine would check all of the following strings against the pattern.

1. Financial Report</b
2. Financial Report</
3. Financial Report</
4. Financial Report<
5. Etc…
20.

While this example shows 20 different strings being looked at, imagine if your regex method were attempting to process an entire HTML file.

Lazy

A lazy quantifier operates using the opposite approach. It starts at the beginning of the text looking for matches. We can make any curly bracket quantifier lazy by following it with a question mark. So if we take our pattern **<.{0,}>** and change it to **<.{0,}?>** we get a different result, in this case: Financial Report. The regex engine found the **<**, the letter **b**, and the **>**, so it found a match. Since we used the global option, the engine kept reading, and eventually found another **<**, followed by 2 characters **/b**, followed by the **>** sign.

Possessive

A possessive quantifier operates just like a greedy quantifier, except that it doesn't back track. If the entire string matches the pattern, a match is returned. If not, the match fails. You can add a + to the end of a quantifier to make it a Possessive quantifier. However, the Microsoft regex engine does not support possessive quantifiers. However, you can instruct the regex group to not backtrack, making it operate like a possessive quantifier.

Possessive quantifiers offer a substantial speed improvement over greedy quantifiers when the match fails. A greedy quantifier says, “If at first you don’t succeed, try, try again.” A possessive quantifier says, “If at first you don’t succeed, give up, no sense being a fool about it.”

Quantifier Symbols

While you can use the {} symbols to specify the number of matches, there are three commonly used quantifier symbols as well. Remember by default, they are greedy, trying to match as much as possible.

Table 6: Common Quantifiers

Quantifier	Matches	Same as
*	Match zero or more times	{0,}
+	Match one or more times	{1,}
?	Match zero or one time	{0,1}

To make any of the above lazy, add the question mark immediately after the quantifier. To review our example, let’s look at how the Financial Reports string will be matched. This assumes global, i.e. all possible matches are returned.

Table 7: Financial Reports String

Pattern	Result
<.*>	Financial Report
<.+>	Financial Report
<.?>	Financial Report
<.*?>	Financial Report
<.+?>	Financial Report
<.??>	Financial Report

Notice that using the question mark quantifier (same as {0,1}) will not match the final tag since it contains two characters between the < and > characters.

Note: Be careful with the * quantifier. \d* might look like it should match any digit, but it will match everything, since pattern of 0 digits will always be found in a search string. The match object will return the Success property of true, but the Value property will be an empty string and Index and Length will both be zero.

Summary

In this chapter, we covered the quantifier characters to allow you to search for strings of varying lengths. Table 8 below summarizes the quantifiers.

Table 8: Quantifiers

Character	Description
{n}	Preceding patterns occurs EXACTLY n times
{n,}	At least n times, but no upper limit
{n,m}	Between n and m times
*****	Match zero or more times, sames as {0,}
+	Match one or more times, same as {1,}
?	Match zero or one time, same as {0,1}

Chapter 5 Anchors

One of the problems we might run into with simple character sets and quantifiers in a regex, is that the search string might be embedded with a larger string, and we might not want this. For example, if we were to use **'com'** to search for an internet URL extension, it would also find comedy, comic, become, etc.

Let's assume we got this message from our help system.

Hey Suzie,

I called March 26th about Internet issues on hulu.com and ebay.com. I can't connect to either one, so I am missing my favorite sitcoms. I also want to sell some of my comic books, but I can't get on eBay either. Call me at (619) 555-1212.

Thanks

Tom

If we asked the regex engine to find the pattern **com** (global and case-insensitive), we might get the result below:

I called March 26th about Internet issues on hulu.com and ebay.com. I can't connect to either one, so I am missing my favorite sitcoms. I also want to sell some of my comic books, but I can't get on eBay either.

That's not quite what we were expecting.

Anchor Metacharacters

An anchor metacharacter allows us to restrict what matches are acceptable, based on the string position and any characters around the string.

Start and End String Anchors

For example, the **^** (when outside of square brackets) and the **$** are anchor metacharacters with special meaning.

- ^ - Start of string
- $ - End of string

Although our examples so far have dealt with searching a larger string, often regex patterns can be used to validate that a single string looks like the pattern we are looking for. For example, if we designed an input form asking for the user's social security number, we could use the **`^\d{3}-\d{2}-\d{4}$`** pattern to not only ensure what they entered looks like a valid SSN, but also nothing else was entered in the text field.

By default, the ^ and $ anchors refer to the first and last positions in a string; however, the regex option multi-line changes that behavior, so that the ^ and $ anchors refer to the beginning and ending of a line, instead of the entire string.

Note: Anchor characters do not take up position in the string, they simply indicate to the regex engine what is expected around the search string.

Finding the last word

In our help system, the last word from the message is fairly likely to be the person's name, so we can use the following regex pattern to find the last word from the text, **`\w{1,}\s{0,}$`**. Note that the pattern says to find the last word of any length **`\w{1,}`** followed by any number of whitespace characters. Often, the final character of a string is a line-feed, even if not visible in the text. So the expression **`\w{1,}$`** might appear valid, but won't match if any whitespace exists after the final word.

Finding the first word

Finding the first word in a string is simpler, using the pattern **`^\w{1,}`** which says look for any number of word characters starting at the beginning of the string. If you use this in conjunction with the multi-line regex option, you can find the beginning word of every line.

Tip: Some flavors of regex (including Microsoft .NET) support the \A and \Z anchors, beginning and ending of string regardless of multi-line setting. However, JavaScript and some other regex flavors do not support these anchors.

Word Anchors

Another very common anchor is the **`\b`** metacharacter, which is used to indicate a word boundary. In our previous example, if we searched for **`\bcom\b`** rather than just **`com`**, we would find only the com in the hulu.com and ebay.com websites. You will often see this referred to as a whole word search. The \b does not take up a character position in the search string; it is only used to "look around" the search text.

Word anchors are the first character in a string, the last character in the string, or between two characters where one character is a word character (**`\w`**) and the next character isn't (**`\W`**).

Anchors do not need to be used in pairs; for example, using the pattern **\bcom** would find any reference to **com** by itself or at the beginning of a word. So **.com** and **comic** would match, but **sitcoms** would not.

You can use the **\B** anchors to find embedded patterns within strings. For example, using **\Bcom\B** would only find **sitcoms** in our example text, since it is the only occurrence of **com** that has a valid word character or either side, even though those word characters are not considered part of the found result.

Caveat

Anchors are different that using other characters to help isolate words in a subtle way. Referring to our sample text from above:

I called Mar 26th about Internet issues on hulu.com and ebay.com. I can't connect to either one, so I am missing my favorite sitcoms. I also want to sell some of my comic books, but I can't get on eBay either.

If we use the pattern **\bcom\b**, we will find the com domain name for hulu.com and ebay.com, but the delimiting characters (dot at the beginning and either whitespace or dot at the end) are not part of the result.

If we use the pattern **\Wcom\W** we will find the same two locations, except that this time, the dot and space will be included in the result string. While this distinction might not be important for searching, it can become an issue in later chapters as we try to manipulate the values returned by the regex engine.

Summary

In this chapter, we covered the anchor metacharacters. While they are not part of the search string, they are useful to ensure you find the result in the location you are expecting it. Table 9 shows the anchor characters.

Table 9: Anchor Characters

Character	Description
^	Beginning of string (when outside [])
$	End of string or line (depending on options)
\b	Word anchor
\B	Must have valid word around match

Chapter 6 Alternation

Alternation is the ability in a regex to match one of a list of choices. For example, if we want to write a regex to match a URL, we might tackle it as shown in Table 10:

Table 10: Alternation Rules

English rule	Regex pattern
Begins with www	`www`
Need a period	`\.`
Then any number of word characters	`\w{1,}`
Another period	`\.`
One of the following TLDs (net, com, org, edu, info)	`(com|net|org|edu|info)`

With the proliferation of top level domains (TLDs), the English rules above are limited, but the concept of alternation is illustrated. Alternation provides a list of alternatives to consider, enclosed in parenthesis and separated by the pipe character |. The choices in the list do not have to be the same size.

Alternation ()

Alternation provides a list of alternative choices that the search text needs to contain one of. In its simplest form, the list is simply a list of words to match. If we wanted to expand the URL pattern search to include FTP sites as well as websites, we could replace the `www` in the pattern with `(www|ftp)`.

In addition, the items between the pipe characters do not simply have to be words; they can be patterns to search for. Imagine we want a regex pattern to find our assigned help desk person, who we know can be John, Sue, or Bill. One approach to find the technician might be to list all of the variations of their names: `(John|Jon|Sue|Susan|Bill|Will)`. While such an approach can work, we can also use patterns to improve the likelihood of a match.

Patterns in Alternation

As a simple example, let's find John, Jon, Sue, Susie, Suzie, Bill, or Will.

Table 11: Alternation Patterns

English rule	Regex pattern
John or Jon	`Joh{0,1}n`
OR	`|`
Sue	`Sue`
OR	`|`
Susie or Suzie	`Su[sz]ie`
OR	`|`
Will or Bill	`[WB]ill`

The regex alternation pattern becomes `Joh{0,1}n|Sue|Su[sz]ie|[BW]ill`. While the ability to add patterns in addition to literal strings is very powerful, it also can introduce some subtle issues. Looking at the URL pattern we wrote at the beginning of this chapter:

```
www\.\w{1,}\.(net|com|org|edu).
```

We could have written it without the parentheses: `www\.\w{1,}\.net|com|org|edu`. However, this would be interpreted differently by the regex engine. Without the parentheses, this would be interpreted as the following:

Find a string that begins with **www**, followed by a period, any number of letters, another period, and the word **net** OR the word **com** OR the word **org** OR the word **edu**. So the following would be the results, probably not what we were expecting.

- www.facebook.com
- www.yahoo.net
- comic books
- comedy

By adding the parentheses, the regex `www\.\w{1,}\.(net|com|org|edu)` returns the expected result, i.e. the two websites.

- www.facebook.com
- www.yahoo.net
- comic books
- comedy

Note: Using parentheses in an expression creates a group, which we will cover in more detail in a later chapter. Groups come in handy when we want to use a regular

expression to break a larger string to components and manipulate the components. We've been focusing on searching with regex in these first few chapters, where the group is less important.

Resolving Alternation

One caveat when working with alternation is to make sure your choices can be reached. Alternation will generally be resolved left to right. If some of the leftmost patterns are always matched, the alternation will stop rather than looking for a better match later in the list. For example, if we wrote the following pattern:

```
\d*|[A-Fa-F]\*\d*|0X\d*
```

It might look like a valid pattern to find either a numeric value or a hex number. However, the \d* at the beginning means 0 or more occurrences of a number. Any string at all will match zero occurrences of a number, so this pattern will always resolve the left-most item in the list.

Summary

Alternation lets you choose a match among a delimited list of patterns. The alternation metacharacter is simply the | (vertical pipe) to separate a list of patterns. Sometimes, you'll need to wrap your alternation syntax in parentheses to make sure your logic intent is clear.

Chapter 7 Searching Summary

In the previous few chapters, we covered a number of the regex searching patterns to find strings in text, and in this chapter, we will provide a few examples of how to use those options to solve some text processing problems.

Help System Parser

In the beginning of the book, we described the help system we wanted to process messages from. We can now use our various regex patterns to create an object to process those help messages. The purpose of the object is to take a text message and fill the following properties if they can be extracted:

- AssignedTo: Help desk rep who took the message.
- UserDate: Date the user indicated they called.
- UserTime: Time of the call.
- EmailAddr: Email address, if one is found.
- PhoneNumber: Phone number, if found.
- ComplainURL: Website the user complained about.

To create the parse class, we first need to create the class and declare a few variables.

```
public class HelpDeskParser
{
     private Dictionary<string, string> RegExprList = new Dictionary<string,
string>();

     public string AssignedTo;
     public string UserDate;
     public string UserTime;
     public string EmailAddr;
     public string PhoneNumber;
     public string ComplainURL;
```

We are keeping the class very simple, using public variables rather than properties. In our constructor, we are going to load our regex expression list. It is often a good idea to keep the regular expressions loaded in one spot, in this case in a dictionary, to make it easier to update them as testing progresses or to enhance the expressions.

```
public HelpDeskParser()
{
    // Load the patterns into the RegExprList
            RegExprList.Add("ASSIGNEDTO",
"(Joh{0,1}n|Sue|Su[sz]ie|[BW]ill)");
```

```
            RegExprList.Add("USERDATE",
@"((sun(day)?|mon(day)?|tue(sday)?|wed(nesday)?|thu(rsday)?|fri(day)?|sat(urd
ay)?),?\s)?((jan(uary)?|feb(ruary)?|mar(ch)?|apr(il)?|may|jun(e)?|jul(y)?|aug
(ust)?|sep(tember)?|oct(ober)?|nov(ember)?|dec(ember)?)\s)+((0?[1-9]|[1-2][0-
9]|3[0-1]),?(\s|st|nd|rd|th))+([1-2][0-9][0-9][0-9]){0,1}");
            RegExprList.Add("USERTIME", @"\b([1-9]|[0-2]|0[1-9])((:[0-5][0-
9]){0,1}\s{0,}[aApP][mM])\b");
            RegExprList.Add("EMAILADDRESS",@"\b([a-zA-Z0-9_\-\.]+)@((\[[0-
9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\.)+))([a-zA-Z]{2,4}|[0-
9]{1,3})(\]?)\b");
            RegExprList.Add("PHONENUMBER", @"\b((\(\d{3}\) ?)|(\d{3}-
))?\d{3}-\d{4}\b");
            RegExprList.Add("COMPLAINURL",
@"(www\.){0,1}\w{1,}\.(net|com|org|edu|info) ");
        }
```

And finally, we include a method to take the message string and extract the pieces it can get from the message. There is not a lot of code in the method, and the power of the regex engine is on display here.

```
    public bool ExtractHelpDeskInfo(string msg)
       {
            // Clear out previous values.
            AssignedTo  = "";
            UserDate    = "";
            UserTime    = "";
            EmailAddr   = "";
            PhoneNumber = "";
            ComplainURL = "";

            foreach (KeyValuePair<string, string> entry in RegExprList)
            {
                Match theMatch = Regex.Match(msg, entry.Value,
RegexOptions.IgnoreCase);
                if (theMatch.Success)
                   {
                   switch (entry.Key)
                       {
                         case "ASSIGNEDTO": { AssignedTo = theMatch.Value;
break; };
                         case "USERDATE": { UserDate = theMatch.Value; break;
};
                         case "USERTIME": { UserTime = theMatch.Value; break;
};
                         case "EMAILADDRESS" : { EmailAddr = theMatch.Value ;
break ; };
                         case "PHONENUMBER": { PhoneNumber = theMatch.Value ;
break ; };
                         case "COMPLAINURL": { ComplainURL = theMatch.Value;
```

```
break; };
                        default:
                           break;
                     }
                }
            }
            return true;
        }
```

While there are improvements we can make (and will in later chapters), this simple class illustrates how powerful regular expressions can be when dealing with a larger amount of unstructured text.

Named Entity Parsing

Natural Language Processing (NLP) is a part of artificial intelligence that deals with having a computer appear to understand native language text, such as English. The complexities of the English language make this an extremely difficult task for computers to handle. One component of NLP is trying to extract "named entities" from text; named entities could be phone numbers, currency amounts, city and state names, people, etc. This is done with lists, grammar rules, and often with some regular expressions.

For example, if the following ad were found online, the highlighted text might be the named entities we want to extract: the year, the type of car, and the price.

2011 Honda Civic excellent condition color BLUE 56K miles for questions ask for George 215 555 1212 from 8AM TO 5PM THANK YOU... Firm $10,300.

To find the type of car, you would most likely have a list of car makes and models and do a simple string search for these. However, the year and price could easily be handled using regex values.

By combining the list lookup and the regex patterns, writing a program to "read" online classified ads and report on year, type of car, and price would not be very difficult.

An example **CarSearch** class is illustrated below:

```
    public class CarSearch
    {
        private string[] CarMakes = { "ford", "honda", "toyota", "pontiac" };
        public string CarMake;
        public int? Year;
        public double? Price;

        private Regex FindPrice = new Regex(@"\$\d{3,5}|\$\d{1,2},\d{3}");
        private Regex FindYear = new Regex(@"20\d\d|'\d\d");

        public Boolean ReadAd(string AdText)
```

```
        {
            Boolean FoundCar = false;
            CarMake = "";
            Year = null;
            Price = null;

            // Search for a car make in Ad text.
            foreach (string Car in CarMakes)
            {
                string CarPattern = @"\b" + @Car + @"\b";
                if (Regex.IsMatch(AdText, CarPattern,
RegexOptions.IgnoreCase))
                {
                    CarMake = Car;
                    FoundCar = true;
                    break;
                }
            }
            // Try to get year and price.
            if (FoundCar)
            {
                // Can we find a price?
                Match PriceMatch = FindPrice.Match(AdText);
                if (PriceMatch.Success)
                {
                    Price =
Convert.ToDouble(PriceMatch.Value.Replace(",","").Replace("$",""));
                }
                // Can we find the year?
                Match YearMatch = FindYear.Match(AdText);
                if (YearMatch.Success)
                {
                    Year = Convert.ToInt16(YearMatch.Value.Replace("'",
"20"));
                }
            }
            return FoundCar;
        }
    }
```

The class provides a method called **`ReadAd()`** which takes the ad content as a parameter. It will search for one of several models, and if the model is found, will attempt to return the price and year (as long as it is greater than 2000) of the car it found.

Note: The ? after the INT and DOUBLE type declarations allow us to place a NULL value in those fields, which is more appropriate if we don't find a match in the message text. It is a shortcut for `Nullable<int>` or `Nullable<double>`.

Validation

One caveat with regular expressions is that they will sometimes find patterns that look alright, but might not be accurate. While regex searching by itself is a very useful tool, you should also look further to see if the returned string is valid, not just that it matches a pattern. Let's look at some examples.

Credit Cards

The following regex will identify most credit cards:

```
^((4\d{3})|(5[1-5]\d{2}))(-?|\040?)(\d{4}(-
?|\040?)){3}|^(3[4,7]\d{2})(-?|\040?)\d{6}(-?|\040?)\d{5}
```

However, even values that match the pattern might not be a valid credit card number. But credit card numbers have a checksum digit built (using the Luhn algorithm) into the number that ensures not only the pattern looks OK, but that it is also a valid combination of numbers. The following C# function performs the checksum function and returns a Boolean value indicating whether or not the credit card number is valid.

```
public bool CC_Check(string CCcard)
{
     // Create an array of digits from the credit card
     CCcard = CCcard.Replace("-", "");
     int[] digits = new int[CCcard.Length];
     for (int len = 0; len < CCcard.Length; len++)
         {
             digits[len] = Int32.Parse(CCcard.Substring(len, 1));
         }
     //Luhn Algorithm
     int RunningTotal = 0;
     bool IsEven = false;
     int nextDigit;
     for (int i = digits.Length - 1; i >= 0; i--)
         {
            nextDigit = digits[i];
            if (IsEven)
               {
                   nextDigit *= 2;
                   if (nextDigit > 9)   {  nextDigit -= 9; }
               }
               RunningTotal += nextDigit;
               IsEven = !IsEven;
          }
         // Must be divisible by 10
         return RunningTotal % 10 == 0;
}
```

If you need to validate credit card numbers, use the regex in combination with the CC_Check() function to ensure you've got a valid card number.

Note: It is still possible after both checks to have a valid, but unissued credit card number, but such a function would require some sort of API communication with the credit card issuer.

EAN 8 number

The EAN 8 number is a simple 8 digit number typically used with a barcode for smaller products. The regex is valid and the EAN-8 number is simply 8 digits, **\d{8}**. This is a primary example where a regular expression would very likely find more matches than only EAN-8 bar codes. Fortunately, the last digit is a checksum; the code below is a C# function to validate that checksum.

```
        public bool EAN8_check(string EAN8)
        {
            // Create an array of digits from the code.
            int[] digits = new int[8];
            int theSum = 0;
            int lastDigit;
            for (int x = 0; x <= 7; x++)
            {
                digits[x] = Int32.Parse(EAN8.Substring(x, 1));
                if ((x % 2) == 0)
                {
                    digits[x] *= 3;
                }
                if (x < 7) { theSum += digits[x]; }
            }
            lastDigit = digits[7];
            return ((theSum + lastDigit) % 10 == 0);
        }
```

Email

Email is often validated using regular expressions; a quick search for regex email validation will yield dozens of email validation regex patterns. However, most patterns are either too broad (i.e. **(\w[-._\w]*\w@\w[-._\w]*\w\.\w{2,3})**) or will prevent some valid email patterns from getting through. For example, many expressions don't accept a + sign in the email address, even though it is a valid character. Email addresses are based on RFC (Request for Comment) 5321 (and others), so a true regex pattern to validate email should handle that standard. You can view such a pattern at the website here. However, the pattern is over 6,000 bytes long.

Note: The example email regex pattern above does not work in .NET, only Perl, but illustrates how complex a pattern to validate email can become.

If you are creating an application that requires email validation, you should visit Dominic Sayer's website, which provides a great email validator that handles email addresses that many regex patterns don't. Of course, it would probably be better to accept an email address from a potential customer rather than display an error message for an invalid email address and lose that customer.

Summary

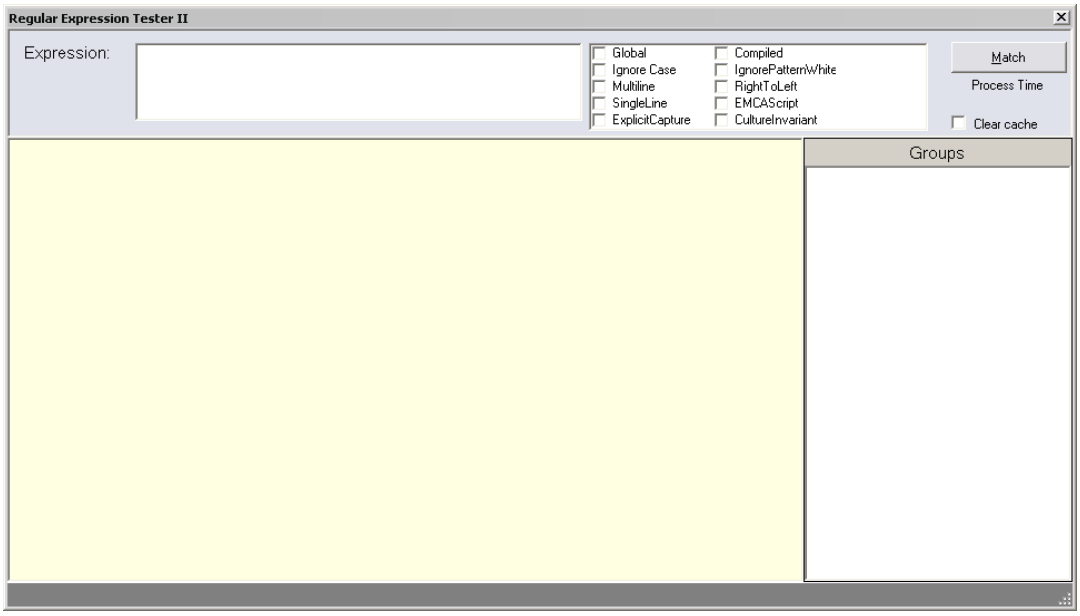
In the first half of this book, we focused on how to search with regular expressions and covered a fair amount of what can be done with the regex literal characters and metacharacters. It is important to keep in mind that while regex is a powerful tool for searching text, you should also consider additional validations, such as checksum, API lookups, etc. You can use the regex to determine whether or not a string is worth running a checksum against or sending to an API, but don't rely on it for strict validation that the data is OK.

In the next section of the book, we are going to explore some additional regex features, and while we continue to focus on searching, we will also begin to explore what can be done with the resulting string or group(s) that a regex finds. For example, we might to want some code to not only extract phone numbers, but to also slice the phone number into area code, main number, and possible extension. So if you feel comfortable with using regex patterns for searching, read on to see what else can be done with them.

Chapter 8 Regex Tester Program II

In order to explore some of the features in the next few chapters, we need to improve our regex tester program by adding a few more features. While the initial program was good for testing searching patterns out, we are going to explore groups and captures, look-arounds, and optimization in the next few chapters, so we need to create a new and enhanced version of our tester application.

You can download the complete application here. Be sure to either download the application or create it yourself to help explore the remaining chapters in the book.



Regular Expressions Tester

Timing

One of the features we are adding is the ability to report how long the engine takes to process the regular expression. We will use the Stopwatch class found in the **System.Diagnostics** assembly. This allows us to report the number of milliseconds an expression takes to process.

```
        private void RunBtn_Click(object sender, EventArgs e)
        {
            Stopwatch stopWatch = new Stopwatch();  // Create a high
resolution time.
            // Start the timer.
            stopWatch.Start();
            // Process the expression.
```

```
            // Record how long the processing took.
            stopWatch.Stop();
            TimeSpan HowLong = stopWatch.Elapsed;
            Double TotalTicks = HowLong.TotalMilliseconds;
            TimeLabel.Text = TotalTicks.ToString("F3") + " ms";
        }
```

This will allow us to try out various optimizations to make our regular expressions run quicker. For most of the examples we've looked at so far, you will probably not notice a difference. However, if you are processing a large number of large text inputs, squeezing a few extra milliseconds can add up.

Clearing the Cache

We've also added a check box to clear the regex cache. This will allow you to test the impact of compiling the regex in advance, which is a particularly useful optimization for regex patterns that will be used repeatedly in your code. The code to clear the cache is shown below:

```
            if (clearCache.Checked)
            {
                Regex.CacheSize=0;
                clearCache.Checked = false;
            }
```

Groups

In the next chapter, we will be exploring how groups work in regular expressions. By adding a tree view component to our testing application, we can see the groups that are found when processing the expression. The code to populate the tree view with groups and captures is shown below:

```
    rootNode.Add(TV.Nodes.Add("[ "+theMatch.Value+" ]"));
    TreeIndex++;

    foreach (string groupName in theExpr.GetGroupNames())
         {
             Group theGroup = theMatch.Groups[groupName];
             if (groupName != "0")
                {
                  TreeNode ChildNode = rootNode[TreeIndex].Nodes.Add("<" +
groupName + "> (" + theGroup + ")");
                  foreach (Capture theCapture in theGroup.Captures)
                       {
                          ChildNode.Nodes.Add(theCapture.Value);
                       }
```

```
                }
            }
```

All Options

Finally, we are going to provide the check box for all regular expression options, not just the subset that we introduced in earlier chapters.

```
private void RunBtn_Click(object sender, EventArgs e)
{
  Boolean isGlobal = false;
  RegexOptions theOpts = RegexOptions.None;

  foreach (var item in CB.CheckedItems)
  {
    if (item.ToString().ToLower() == "global") { isGlobal = true; }
    if (item.ToString().ToLower() == "multiline")
           { theOpts = theOpts | RegexOptions.Multiline; }
    if (item.ToString().ToLower() == "ignore case")
          { theOpts = theOpts | RegexOptions.IgnoreCase; }
    if (item.ToString().ToLower() == "singleline")
          { theOpts = theOpts | RegexOptions.Singleline; }
    if (item.ToString().ToLower() == "compiled")
          { theOpts = theOpts | RegexOptions.Compiled; }
    if (item.ToString().ToLower() == "ignorepatternwhitespace")
          { theOpts = theOpts | RegexOptions.IgnorePatternWhitespace; }
    if (item.ToString().ToLower() == "righttoleft")
          { theOpts = theOpts | RegexOptions.RightToLeft;}
    if (item.ToString().ToLower() == "emcascript")
         { theOpts = theOpts | RegexOptions.ECMAScript; }
    if (item.ToString().ToLower() == "cultureinvariant")
         { theOpts = theOpts | RegexOptions.CultureInvariant; }
  }
```

Match button

When the match button is clicked, the following code is performed after the options are set:

```
            // Start the timer.
            stopWatch.Start();

            List<TreeNode> rootNode = new List<TreeNode>();
            int TreeIndex = -1;

            // If global, then iterate the Matches collection.
```

```
            if (isGlobal)
            {
                try
                {
                    Regex theExpr = new Regex(pattern,theOpts);

                    foreach (Match match in theExpr.Matches(source))
                    {
                        int endindex = match.Length;
                        rtb.Select(match.Index, endindex);
                        rtb.SelectionBackColor = BGHighlight;
                        rtb.SelectionColor = FGHighlight;

                        rootNode.Add(TV.Nodes.Add("[ " + match.Value + "
]"));
                        TreeIndex++;


                        foreach (string groupName in theExpr.GetGroupNames())
                        {
                            Group theGroup = match.Groups[groupName];
                            if (groupName != "0")
                            {
                                TreeNode ChildNode =
rootNode[TreeIndex].Nodes.Add("<" + groupName + "> (" + theGroup + ")");
                                foreach (Capture theCapture in
theGroup.Captures)
                                {
                                    ChildNode.Nodes.Add(theCapture.Value);
                                }
                            }
                        }
                    }
                }
                catch (Exception ex)
                {
                     SLAB.Text = ex.Message;
                }
            }
            else
            {
                try
                {
                    Regex theExpr = new Regex(pattern, theOpts);
                    Match theMatch = theExpr.Match(source);
                    if (theMatch.Success)
                    {
                        int endindex = theMatch.Length;
                        rtb.Select(theMatch.Index, endindex);
                        rtb.SelectionBackColor = BGHighlight;
```

```
                        rtb.SelectionColor = FGHighlight;

                        rootNode.Add(TV.Nodes.Add("[ "+theMatch.Value+" ]"));
                        TreeIndex++;

                        foreach (string groupName in theExpr.GetGroupNames())
                        {
                            Group theGroup = theMatch.Groups[groupName];
                            if (groupName != "0")
                            {
                                TreeNode ChildNode =
rootNode[TreeIndex].Nodes.Add("<" + groupName + "> (" + theGroup + ")");
                                foreach (Capture theCapture in
theGroup.Captures)
                                {
                                    ChildNode.Nodes.Add(theCapture.Value);
                                }
                            }
                        }
                    }
                    else
                    {
                        SLAB.Text = "Not found...";
                    }
                }
                catch (Exception ex)
                    {
                        SLAB.Text = ex.Message;
                    }

            }
            TV.ExpandAll();
```

One of the changes we needed to make was to create a regex object rather than rely on the static regex class. This is because certain group functionalities, such as GetGroupnames(), are not available as a static regex method.

Chapter 9 Regex Objects

In this chapter, we are going to review the various regular expression objects that are returned by the regex match calls. A lot of details are available in these object about where the match was located, groups, etc.

Match object

The match object (which we briefly discussed in Chapter 2) is returned directly from the Regex Match() method, or the Regex Matches() method returns a collection in which each item in the collection is a match object. You can use the properties to explore the information collected during the match process.

Properties

The following properties return information about the result of the entire expression:

- Index: The index property returns the zero-based position in the string where the match was found.
- Length: The length property returns the length of the result captured by the expression.
- Success: This is a Boolean value indicating whether or not a match was found.
- Value: The value property is the actual result string that the expression found.
- Groups Collections: Returns a collection of individual group objects the regular expression engine found. If you don't specify any groups, this collection can be empty. Otherwise, it will contain a list of group objects found while processing the expression.

Captures Collections

When the regular expression engine parses the string, it will capture each element of text as it processes, looking for matches. Most regex engines discard the match as it continues searching in this process, but the Microsoft engine saves these discarded matches in a collection called Captures. To illustrate, let's consider the following regex to get a list of words in a sentence: **\b(\w+\s*)+**

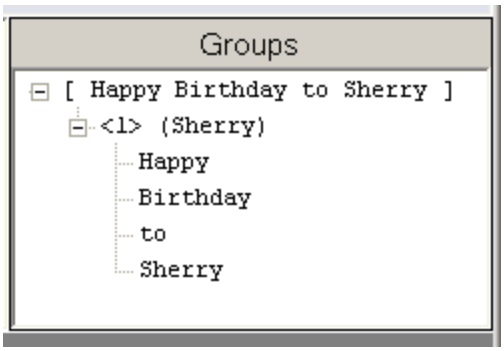
Table 12: Example Regex

English rule	Regex pattern
Begin at a word boundary	**\b**
Start a group	**(**
Within group, find word	**\w+**

English rule	Regex pattern
Followed by any number of spaces	`\s*`
Close the group	`)`
Group can occur 1 or more times	`+`

Each word in the sentence would be considered a match; however, only the last word in the sentence gets assigned to the group. While other words will be discarded as non-matches, you can find them in the captures collection when using Microsoft's regex object.

If I wish my friend Sherry happy birthday using the above pattern, I'll get the following tree structure:



Tree Structure

The first group is the last word in the sentence **<1> *Sherry*** and the captures (discarded matches) are each word in the sentence, including the final word that was assigned to the group.

We will explore groups and captures in much greater detail in the next chapter.

Group Object

The group object contains similar properties to the match object, providing details about the sub-expression search results.

Properties

The following properties return information about the result of the sub-expression.

- Index: Returns the zero based position in the string where the match was found.
- Length: Returns the length of the result captured by the sub-expression.
- Success: This is a Boolean value indicating whether or not a match was found.

- Value: The actual result string that the sub-expression found.
- Captures Collection: Each group sub-expression will also contain a collection of the captured search results from the sub-expression defined in the group.

Capture Object

The capture object contains basic information about each search text result. It can be obtained from the match object's capture collection or the group object's capture collection. It allows you to get a hold of all texts returned for either the entire expression or the individual group sub-expressions.

Properties

The following properties return information about the search texts that were evaluated during the expression search process. Note that in a group scenario, the last evaluated search text is returned in the group object and also present in the capture collection.

- Index: Returns the zero based position in the string where the match was found.
- Length: Returns the length of the result captured by the sub-expression.
- Value: Value contains the actual result string that the sub-expression found.

Chapter 10 Groups

In the previous chapters, we explored searching with regex, but did not do very much with the results; we simply checked to see if the conditions were met, and possibly used the result to illustrate in the RTF control where the text was found. In the next few chapters, we are going to focus on using the results for further processing, such as data cleaning, text replacements, etc.

Regex Groups

A group in a regular expression is an embedded regular expression within a larger one. You can name the groups or refer to them by numeric position in a collection. By using groups, you can perform multiple operations on a string and gather multiple results. You can also refer to other groups from within the expression, to search for duplicates.

Capturing Groups

To create a group to capture a pattern, you must enclose a regular expression within parentheses. This will create a subexpression pattern within the overall pattern. The subexpression can be any valid regular expression pattern inside the parentheses. For example, if we take a very simple phone number pattern of **`\({0,1}\d{3}\){0,1}[/\\ -]\d{3}-\d{4}`**, it will match phone numbers of the following forms:

- 610-555-1212
- (215) 555-1212
- 609\555-1212

However, our requirement is to split the phone number into area code and phone number. While the pattern will detect valid numbers, we won't know what the area code and the phone number is. By using groups, we can both search for phone numbers and split them into two component pieces.

```
\({0,1}(\d{3})\){0,1}[/\\ -](\d{3}-\d{4})
```

By placing parentheses around the area code and the phone number patterns, we now not only get valid phone number, but the phone number is broken into two pieces, one for each sub pattern. Each component piece is referred to as a group and the groups are numbered from left to right, starting at the index of 1.

Note: Regex groups start at the index of 1, rather than 0 as most .NET collections do; keep this in mind when working with regex.

When we now process this regex (using Match to obtain the match object), the match object will contain a collection property called Groups. In this example, there will be two groups: Group 1 containing the area code digits, and Group 2 containing the actual phone number.

Naming Groups

The Microsoft .NET framework (and other regex engines) allows you to assign names to the groups within a sub-expression. To assign a name to a group, you can use either of the following two methods:

- `(?<name>regular expression)`
- `(?'name'regular expression)`

You can refer to the group by name in the expression using the `\k<name>` or `\k'name'` syntax. We can add names to our phone number expression, as shown below:

```
\({0,1}(?<areacode>\d{3})\){0,1}[/\\ -](?<number>\d{3}-\d{4})
```

When using the named approach, the group property of the match object can be referenced by either name (case-sensitive) or index number. In the example above, Group[1] and Group["areacode"] both refer to the same capturing group. One caveat though, any unnamed group will be before the named group, so if we used the following pattern:

```
\({0,1}(?<areacode>\d{3})\){0,1}[/\\ -](\d{3}-\d{4})
```

The phone number would be Group 1, the group named areacode would be Group 2. To avoid this problem, I would recommend being consistent with group naming; you should either name all groups or none. Group names can make your regular expression more readable, particularly when most regex engines support up to 99 groups with a pattern.

Note: Naming of groups is one area that regex engines often vary in. Python's regex engine uses the ?P<name> syntax to name groups and the ?P=name to reference them. Python was the first to offer named groups, but no standard was set, leaving .NET engineers to define their own syntax for group names.

Multiple Groups

You can create a regex using groups and quantifiers; however, the regex engine will only put the last match into the group. For example, if we use the following expression, we are asking to get the function name and all parameters from a SQL function call (for simplicity, removed the parameter data types).

```
(?<Method>\w*\()(?Parameter>@\w*[,)])+
```

If we run our code against this example function:

```
Function GetName(@ObjId,@runDate) as INT
BEGIN

END
```

We will get two groups back. The first one, called <Method>, will contain **GetName(** and the second group, <Parameter>, will contain **@runDate)**. Only the last parameter will be assigned to the group.

Fortunately, the Microsoft regex will also save all the texts that met the search pattern. While only the last one will be put in the group object, the others will be put into the Captures collection for the group.

Tip: The Captures collection is zero-based, while the Groups collection is one-based. Keep this in mind if you need to work with both collections.

Back References

Your regular expression can make references to capturing groups using a back reference. Since each group is numbered starting left to right at one, you can refer to a group by a backslash followed by a group number, or the \k<name> syntax for named groups.

As an example, let's consider how we might write a regex pattern to detect duplicates words in a sentence. To describe this in English, we would say: get the first word, then the next word. If the next word is the same as the previous word, report it.

Table 13: Back References

English rule	Regex pattern
Find a word	(\w+)
A space or comma	[\s,]
The last word we found	(\1)

If we now were to write the following sentences in our test program:

She thought her test was very, very unfair.
However, her professor said the the test was actually pretty easy.

If we then apply our regex pattern of **(\w+)[\s,](\1)**, we would see the following results:

She thought her test was very, very unfair.
However, her professor said the the test was actually pretty easy.

You don't need to make the back reference a group, as the pattern **(\w+)[\s,]\1** would give the same results; however, no information would be available about the second duplicate word in the string.

Group 0

The regex engine has a group 0, which is the entire regex pattern (as though you put parentheses around the entire expression). If you are using groups within your pattern, you probably don't need to reference group 0 at all, but it can be convenient to get the captures evaluated by the entire expression.

Note: In our regex tester program, we hide group 0 from the tree view display, but you can replace the `if (groupName != "0")` with `if (true)` to display the group 0 details and capture information in the treeview.

Non-Existing Groups

If you try to reference a non-existing group, the group object will return a value of false for the **Success** property. You might also have an empty group, a group that was requested in the pattern, but never found in the search string. For example, if we were to validate a phone number, we might want to look for a left parenthesis to start the area code; which would look like the following:

```
(\()?
```

Look for the parenthesis character zero or one times. If the left parenthesis is found, it will be placed in a numeric group. You need to capture the group for later referencing to find out if the parenthesis was found (and hence look for the matching right parenthesis).

Testing for Groups

You can use the regex conditional pattern to test whether a group has a value or not. The pattern is indicated by **(?(group number or name)<pattern>)**. This construct says, if the group is found, apply the pattern indicated. If not, then do nothing. You can also use the | character to act as an else statement, in case you want to use one pattern if the group is found and another pattern if it is not. The general syntax is as follows:

```
(? (group name or number) <true-pattern>|<false-pattern)
```

Forward References

A forward reference is a group reference number that is found in the pattern prior to the group it references. The regex engine can handle forward references as well as backward references.

Non-Capturing Groups

Generally, when you creating sub-patterns using the group construct, you want to do something with the groups. However, it is very possible you need to group for searching purposes, but don't plan on using it. When this occurs, you should use a non-capturing group, which is a group that takes part in the search but cannot be referenced in the pattern. A non-capturing group will not be returned in the regex group collection.

Creating a Non-Capturing Group

To create a non-capturing group, you use the pattern **(?: <pattern>)** syntax. You should create non-capturing groups if you will never reference the group (either through a back reference or in the group collection in the regex object).

There is some memory overhead associated with saving groups, so using a non-capturing group will (very slightly) reduce the memory used by the pattern.

Summary

Groups of sub-patterns within a regex come in very handy to slice out parts of the string and are very useful to replace and reformat strings. You should name groups you need and set the non-capturing option for the groups which are needed for processing, but not needed by your code.

Chapter 11 Look-Arounds

Look-arounds allow a regular expression to find patterns that are either followed or preceded by another pattern. The look-arounds are not actually matches or groups, but simply assertions that the rest of the pattern can be tested for a match. You can use look around to check that patterns are or are not found before or after the pattern you are searching for.

Look-Aheads

Look-aheads are used to test whether your pattern is followed (or not followed) by another pattern. For an example, let's consider that we have a list of cities and state codes. We want to find any city that is in New Jersey, Pennsylvania, or Delaware, and ignore all other states. Our sample data is shown below:

```
Baltimore MD
Trenton, NJ
Philadelphia PA
Rochester, NY
Claymont, DE
```

Positive Look-Ahead

Our English and regex rules are shown below. We are using the positive look-ahead pattern of **(?=<pattern>).**

Table 14: Postive Look-Ahead

English rule	Regex pattern
Start a named group	**(?<city>**
Any word	**\w+**
Close group	**)**
One or more commas or spaces	**[,]+**
Must be followed by NJ, PA, or DE	**(?=NJ|PA|DE**
Close the look-ahead	**)**

If we use the regex pattern **(?<city>\w+)[,]+(?= NJ|PA|DE)** then the following results will return matched. Note that while the comma or space is matched as part of the search, it is not part of the group's collection strings.

```
Baltimore MD
Trenton, NJ
Philadelphia PA
Rochester, NY
Claymont, DE
```

The regex object will create a group containing the city name we found. We could add a second named group to get the state code as well, using the following pattern:

```
(?<city>\w*)[, ]+(?=(?<state>NJ|PA|DE))
```

This type of pattern is called a positive look-ahead. Look for the pattern after the potential match and ensure the pattern is found.

Using Look-Aheads for Password Rules

One approach to understand look-aheads is to consider it a conditional expression. We are saying you can accept this pattern IF and ONLY if you find an additional pattern in the string as well. Let's see how we might use this to ensure a password matches the business rules.

Our password rules are very simple; it must be between 6 and 15 characters, contain one uppercase letter, and one number. Of course, our definition of simple and our user's definition might differ a tad. Since we expect to test the entire string, we will use anchors as well.

We start with a rule to create a password group, only if we find a string of letters between 6 and 15 characters long.

```
^(?<pwd>(?=\w{6,15}$).*)
```

Essentially, our group will match all characters (.*) as long as the condition of length between 6 and 15 characters is met. However, we need additional conditions, so we add another look-ahead pattern to the expression. This one ensures that in addition to being between 6 and 15 characters, at least one digit is found.

```
^(?<pwd>(?=\w{6,15}$)(?=.*?\d).*)
```

Our last rule requires an upper case letter, so we add that look-ahead pattern as well.

```
^(?<pwd>(?=\w{6,15}$)(?=.*?\d)(?=.*?[A-Z]).*)
```

Our final expression says "accept all characters as long as the three look-ahead conditions are satisfied." You can have multiple look-aheads in your pattern to make the search rules as detailed as you need them to be.

Negative Look-Ahead

Now, let's imagine I won the lottery, and decide to buy myself a Lexus. However, I don't want a hybrid, call me eco-unfriendly. So I want to scan the online ads looking for a Lexus car, but I do not want a hybrid. In English terms: find any Lexus model UNLESS it is a hybrid. To build the expression, we use the **(?!<pattern>)** negative look-ahead, as Table 15 shows.

Table 15: Negative Look-Ahead

English rule	Regex pattern
The word Lexus	**Lexus**
Any number of spaces	**\s+**
Start a named group	**(?<model>**
Two letters	**[A-Z]{2}**
Close the group	**)**
Not followed by Hybrid	**(?!Hybrid)**

Now, we find a classified ad with the following text:

I am selling my 2013 Lexus CT Hybrid, stop by for a test drive.

The regex pattern will not find a match in the above text, however, it will for the following ad:

I am selling a 2012 Lexus RX with low mileage and a great price!

This ad will be matched by the pattern **Lexus\s+(?<model>[A-Z]{2})(?!\sHybrid)** and the model group will contain RX.

Back to Our Password

Now, let's revisit our revisit our password regex. Users don't like the uppercase requirement, so we decide to remove that. Our simpler regex pattern is now this:

^(?<pwd>(?=\w{6,15}$)(?=.*?\d).*)

However, we don't want to accept passwords that are too easy to get, so we are going to add some conditional logic to reject certain passwords. We do this with a negative look-ahead. Our new pattern is:

^(?<pwd>(?=\w{6,15}$)(?=.*?\d)(?!abc123|123456|password\d).*)

One pattern now says the password must be between 6 and 15 characters, at least one digit, AND cannot be abc123, or 123456, or the word password followed by a digit.

Look-Behinds

Look-behinds work similarly to look-aheads, except they check for patterns that precede the search pattern. The look-behind syntax is:

(?<= <pattern>) Positive look-behind, pattern must be found before result text

(?<! <pattern>) Negative look-behind, pattern must not be found before result text

Positive Look-Behind

To consider an example, let's consider a JSON file that contains a person's address information.

```
{
    "firstName": Michelle,
    "lastName": Obama,
    "age": 50,
    "address": {
        "streetAddress": 1600 Pennsylvania Ave,
        "city": Washington,
        "state": DC,
        "zipCode": 20500
    },
}
{
    "firstName": Mary,
    "lastName": Bachman,
    "age": 24,
    "address": {
        "streetAddress": 105 Fayette Street,
        "city": Conshohocken,
        "state": PA,
        "zipCode": 19428
    },
}
```

Our application needs to extract everyone who lives in Pennsylvania from this JSON file, so we need to confirm the current record has a state code of PA or the state spelled out. However, we don't simply want to search for the state name, because there is a rather famous building on Pennsylvania Avenue in Washington DC.

The regex pattern to find Pennsylvania residents is simply:

```
(?<state>PA|Pennsylvania)
```

The pattern means find any PA or Pennsylvania and place in a group called **State.** Of course, in this example, the JSON entry for Michelle Obama will also show up, since she lives at Pennsylvania Ave.

In order to assure we find only state codes from the “state” entry in the JSON file, we use a positive look-behind. It is constructed as shown below:

Table 16: Positive Look-Behinds

English rule	Regex pattern
Must find “state”: first	`(?<=\"state\":)`
Now a space	`[ ]`
Any number of characters	`.*`
Start a named group	`(?<state>`
Either PA or Pennsylvania	`PA|Pennsylvania`
Close the group	`)`

Our resulting regex pattern is defined below:

```
(?<=\"state\":)[ ].*(?<state>PA|Pennsylvania)
```

This will find Mary’s record since she lives in the state of PA and will skip Michelle’s record since she lives in DC. We could also add a look-ahead, in this case, if we want to make sure a zip code field can also be found:

```
(?<=\"state\":)[ ].*(?<state>PA|Pennsylvania)[, ].*(?=\"zipCode\")
```

Note: This pattern requires that the Single Line option be utilized, to allow the period character to match the new line character as well.

You can freely combine look-aheads and look-behinds to focus on the exact text you are looking for.

We could use a similar pattern to find the person’s age, matching a two digit number, as long as it was preceded by the keyword “age.”

Negative Look-Behind

A negative look-behind says find a pattern that is not preceded by another pattern. The regex syntax is **(?<!** followed by a pattern and a closing parenthesis. For example, imagine we want to find all websites in a Windows host file, but want to exclude those on comment lines. The sample host file might look like the following:

```
# For any domain names you would rather not see, simply add a line
# that reads "127.0.0.1 machine.domain.tld". This will redirect any
# requests to that host to your own computer.
#<localhost>
127.0.0.1   localhost
127.0.0.1   localhost.localdomain
255.255.255.255   broadcasthost
127.0.0.1   local
127.0.0.1   www.badStuff.net
#</localhost>
```

We now want a simple regex pattern to find IP addresses with the file, unless they are preceded by a # comment marker. The regex pattern would look like this:

```
(?<ip>\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})(?<!#.*)
```

The first group looks for a very simple IP address (it is not limited to integers of 255 as an actual IP address check would need to do). The second (non-matching group) is the code for a negative look-behind which says to match the IP address only if there is a not a comment character, any number of characters, followed by the end of line marker (we use the multi-line option to make the $ represent the end of the line).

Summary

Look-arounds are useful patterns to let you perform conditional logic as part of your search pattern. They only accept a match if one or more other patterns can be found (or not found). They provide a very powerful addition to the regex pattern matching logic.

Chapter 12 Replacing Text with Regex

The Microsoft Regex class provides a **Replace** method to update text based on the results of a regular expression. This method has a number of overloads to provide a great deal of flexibility in replacing text. The Replace() method can be called on a regex object or the Static Regex class (which requires passing the regex pattern as a parameter).

Replace()

The simplest variation of Replace takes a source string and replaces any matching pattern with a replacement string. If we wanted to remove extra spaces from a string (replace all multiple spaces with a single space), we could create a regex pattern like **\s{2,}** which looks for any white space that occurs at least two times. We then use the Replace() methods with two strings.

Replace(sourceString, ReplaceWith)

This is to find all occurrences of the pattern and replace it with the replacement string.

```
Regex ExtraSpaces = new Regex(@"\s{2,}");

string SourceInput = "Ben  &   Jerry's makes great ice cream...";
string TweakedResult = ExtraSpaces.Replace(SourceInput, " ");
```

The original string above was: `Ben  &   Jerry's makes great ice cream...`

And the result string is: `Ben & Jerry's makes great ice cream...`

For another example, remove all dollar signs and commas from a string to make it a numeric value.

```
Regex ExtraSpaces = new Regex(@"[$,]");

string SourceInput = "$12,750.00";
string TweakedResult = ExtraSpaces.Replace(SourceInput, "");

// Returns  12750.00
```

The static method works the same way, except that the second string is the pattern, so the currency cleaner using static methods would look like this:

```
string SourceInput = "$12,750.00";
string TweakedResult = Regex.Replace(@SourceInput, @"[$,]", "");

// Returns  12750.00
```

Using Groups

If your regex pattern creates groups (or sub-patterns), you can reference those groups by either name or number in your replacement text parameter. For example, let's take a simple phone number regex that only handles phone numbers formatted as "xxx-xxx-xxxx." We are going to break the area code and the phone number into separate groups. Our regex expression is as follows:

```
(?<area>\d{3})[- ](?<phone>\d{3}-\d{4})
```

We can create a replacement string to create a formatted phone number which consists of parentheses around the area code, a space, and the seven digit number (with the dash).

```
 string SourceInput = "610-867-5309";
 string TweakedResult = Regex.Replace(@SourceInput,
                        @"(?<area>\d{3})-(?<phone>\d{3}-\d{4})", "($1) $2");

// Returns  (610) 867-5309
```

We can also reference the group names instead of group numbers by using **${group}**. This can make the replacement text a bit more clear.

```
 string SourceInput = "610-867-5309";
 string TweakedResult = Regex.Replace(@SourceInput,
                        @"(?<area>\d{3})-(?<phone>\d{3}-\d{4})",
                        "(${area}) ${phone}");

// Returns  (610) 867-5309
```

This type of replacement comes in very handy when attempting to clean data to some standards. We will provide a few examples in our next chapter.

Other Replace() Parameters

In addition to the simple string examples, there are a number of additional overloaded methods to control the replace operation.

Specifying Regex Options

If you are using the static method for replacement of strings, you can add a fourth parameter, which is the regex option to apply. This allows you to set the options when the replacement is called, rather than when you instantiate the object using the class methods.

Replace(sourceString, Pattern, ReplaceWith, RegexOptions)

Specifying Time-out (.NET 4.5 and Above)

If you are using the static method for replacement of strings, you can add a fifth parameter, which is a timespan object. This allows you to set the duration that the regex is allowed to run. If you plan on allowing your users to create a regex patterns, I would recommend adding the time span to prevent a poorly designed pattern to backtrack itself into oblivion.

```
Replace( sourceString, Pattern,  ReplaceWith, RegexOptions, timeSpan )
```

Controlling the Number of Replacements

When creating the regex object, rather than using the static methods, you can add a third parameter of type integer. This parameter tells the Replace method the maximum number of replacements it is allowed to make.

```
Replace( sourceString,  ReplaceWith, MaxReplacements )
```

Controlling Where to Start Searching

When creating the regex object, you can also add a fourth parameter of type integer. This parameter tells where in the input string the search begins.

```
Replace( sourceString,  ReplaceWith, MaxReplacements, StartAt )
```

Note that in this syntax, you can use -1 to indicate no upper limit on the number of replacements allowed.

Summary

The Replace method allows you to create a new string based on the regex pattern and the replacement string. It is very useful to clean data and create standardized results. In the next chapter, we will use regex and Replace() for the common programmer task of cleaning data.

Chapter 13 Cleaning Data with Regular Expressions

A popular saying among computer professionals is “garbage in, garbage out,” which simply means that the output of a computer application can only be as good as the data put in. People are very flexible when it comes to understanding text in various forms, while computer systems generally work better with structured data. For example, it is easy to recognize the area code portion of a phone number written in any of the following manners:

- 1-215-555-1212
- (610) 867-5309
- 484/ 555-1234
- 203.514.2213

Often, programmers will be asked to “clean up” or normalize data coming into a system. Regular expressions can be a very useful tool in a programmer’s toolbox when faced with such a task.

Phone Numbers

We are going to combine our group pattern with the replace to extract the area code and number from any of the phone number strings and place into a consistent format. Our phone number regex is described below:

Table 17: Phone Number Rules

English rule	Regex pattern
Might begin with a one and some punctuation	`(?:1[-\s.])?`
Possibly a left parenthesis	`(\)?`
The areacode	`(?<areacode>\d{3})`
If group 1 has left parenthesis, get right	`(?(1)\))`
Space, dash, slash, or period	`[-\s.\/]?`
Optional space	`\s?`
Phone number (split into two groups)	`(?<prefix>\d{3})[-\s.]` `(?<number>\d{4})`

The resulting pattern is:

```
(?:1[-\s.])?(\()?(?<areacode>\d{3})(?(1)\))[-
\s.\/]?\s?(?<prefix>\d{3})[-\s.](?<number>\d{4})
```

This pattern will handle each of the phone format examples from above. For any successful match, we will have the area code, the prefix, and the four digit number. We can now apply the replace method to create a consistently formatted phone number. Our goal is to produce the following strings from the previous list of phone numbers:

- (215) 555-1212
- (610) 867-5309
- (484) 555-1234
- (203) 514-2213

Our code will create a regex object and assign the pattern to the object. We then loop through our phone numbers, building replacement, and nicely formatted strings.

```
string[] PhoneNumbers = { "1-215-555-1212", "(610) 867-5309",
                          "484/ 555-1234", "203.514.2213" };
string CleanedPhone = "";

Regex thePhone = new Regex(@"(?:1[-\s.])?(\()?(?<areacode>\d{3})(?(1)\))[-
\s.\/]?\s?(?<prefix>\d{3})[-\s.](?<number>\d{4})");

Regex FinalFormat = new Regex(@"(\d{3}) \d{3}-\d{4}");

foreach (string OnePhone in PhoneNumbers)
   {
      CleanedPhone = thePhone.Replace(OnePhone, "(${areacode}) ${prefix}-
${number}");
      if (CleanedPhone==OnePhone)
         {
           // Nothing changed.
           if (FinalFormat.IsMatch(OnePhone))
              {
                // Phone number is already formatted as expected.
              }
           else
              {
                 // The phone number string we got didn't match the pattern
and is not
                 // already in our final format, so we should probably log
for manual review.
               }
         }
```

```
      else
          {   // Update the phone number database.   }
   }
```

If a phone number looks the same after the replace, it is either already in the correct format (which we check for) or cannot be converted, so we should probably review the output. Most likely, when you run this type of clean-up, you'll get some non-matched input phone numbers. By reviewing the list, and tweaking the pattern, you can probably achieve a pretty high ratio of phone number clean-ups.

People's Names

Names are notoriously difficult to handle, because of the tremendous variety of names. We are going to look at a simple name parser. It will handle common titles, such as Dr, Miss, Mrs, etc. It will also get first names and last names that are either full words or begin with an uppercase letter followed by an apostrophe. This is by no means a comprehensive name check, but it could have a reasonable number of names. The pattern is as follows:

^(?<title>Mrs|Mrs\.|Mr|Mr\.|Miss|Ms|Dr|Dr\.)\s?(?<first>\w+)\s+(?<last>([A-Z]')?\w+)?$

When we process the names, we are going to break the name into three fields and build a SQL update statement to update the record in a MySQL database table. Our sample data looks like this:

- Mr. Joe Booth
- Mr Steve Boatman
- Dr. Peter Deitz
- Dr John Hoffler
- Mrs. Janet Green
- Mrs Jaspreet Kaur
- Ms Kellie Helene
- Miss Mary Bachman
- Mr. Joe D'Angelo

For simplicity, we will load the list of names into an array of strings and build our SQL statement from that list.

```
string[] ListOfNames = {"Mr. Joe Booth",
                        "Mr Steve Boatman",
                        "Dr. Peter Deitz",
                        "Dr John Hoffler",
                        "Mrs. Janet Green",
```

```
                        "Mrs Jaspreet Kaur",
                        "Ms Kellie Helene",
                        "Miss Mary Bachman",
                        "Mr. Joe D'Angelo"
                                    };
StringBuilder sb = new StringBuilder();
Regex NameParse = new
       Regex(@"^(?<title>Mrs|Mrs\.|Mr|Mr\.|Miss|Ms|Dr|Dr\.)
             \s?(?<first>\w+)\s+(?<last>([A-Z]')?\w+)?$",
          RegexOptions.IgnoreCase | RegexOptions.Multiline);

foreach (string FullName in ListOfNames)
   {
    sb.AppendLine(NameParse.Replace(FullName,
        "UPDATE PeopleTable " +
        "SET firstName = `${first}`,LastName=`${last}`,Title=`${title}` " +
        "WHERE SourceName = `$0`"));
   }

// The string builder variable sb now contains the SQL to perform the
updates.
```

Summary

Although regular expressions are a powerful tool, they are not the only tool available. Often, a programmer will be tasked with reading data from a comma separated file (CSV), and regular expressions seem like great way to go, until you start to try it. The rules for the CSV file layout are more complex than they seem at first glance. For example, when should quotes be used? How are embedded commas handled?

Chapter 14 Unicode

Unicode is a computing standard for representing characters in almost every language in the world. There are well over 100,000 characters represented by the Unicode standard. The Unicode standard is a complex but very powerful set of rules to represent characters. For the purpose of this chapter, we are going to look at some very simple Unicode features and how to use regex to search for them.

Code Points

Although a bit more complex in practice, you can think of a code point as a 4-digit hex number representing a character. If you run the regex tester programmer and enter the number 0936 while holding the alt key, you'll get the Greek capital letter psi (Ψ). If you use the regex pattern \u03A8 you'll get a match. The 03A8 is the UTF-16 hex code for the psi character. You can use .NET to determine the hex value using the following code example. Remember, when using the \u pattern, the hex number must be four characters long, including the leading zero.

```
int Dec = 936;
string HexC = Dec.ToString("X");
```

You can also visit the Unicode Character Search site to look up the hex codes for various Unicode code points.

Unicode Categories

Although you may have the need to search for individual characters, you could also search for categories. Each character in Unicode belongs to a category, such as a letter, or a numeric symbol, etc. To specify a Unicode category, you use the **\p{ category }** syntax. The lowercase \p matches Unicode characters within the category, and the upper case \P matches characters that are not in the category. Some common categories are described as follows.

Letter Categories

The \p{L} category represents characters that are considered letters. If you use this regex pattern against the psi symbol from above, it will find a match since psi is considered a letter in Unicode.

You can also distinguish between lower case and upper case letters using the following:

- \p{Ll} - Lowercase letter
- \p{Lu} - Uppercase letter

- \p{Lt} - Title letter (start of word when the first letter is capitalized)

Separator Categories

The \p{Z} category represents characters that are considered separators or white space.

Symbol Categories

The \p{S} category represents symbols, such as math, box-drawing, currency, etc. You can drill down further using the following:

- \p{Sm} - Math symbol
- \p{Sc} - Currency symbol
- \p{So} - Other symbol, not currency or math

Number Categories

The \p{N} category represents numeric symbols, such as digits and things like Roman numerals. You can break these down further using:

- \p{Nd} - Digits zero through nine
- \p{Nl} - Numbers that look like letters, i.e. Roman Numerals
- \p{No} - Superscripts and subscripts, a number outside of 0 through 9, etc.

Punctuation Categories

The \p{P} category represents punctuation symbols, such as dashes, brackets, etc. You can break these down further using:

- \p{Pd} - Dash or hyphen characters
- \p{Ps} - Opening bracket characters
- \p{Pe} - Closing bracket characters
- \p{Pi} - Opening quote characters
- \p{Pf} - Closing quote characters
- \p{Pc} - Connecting, such as underscores that connect words
- \p{Po} - Other punctuation symbols

Unicode Scripts

Each Unicode code point is assigned into a script. A script is the set of code points used in a particular writing system. For example, the psi symbol would be considered part of the Greek script, but there are also Arabic script sets, Cyrillic sets, etc. You can use the \p{ script } syntax to find if a character or word that is part of the script. If we applied \p{IsGreek} to our text of the psi symbol, it would match since psi is a Greek character.

Microsoft refers to these Unicode scripts as Named Blocks and supports a large number of them based on the Unicode 4.0 standard. Some common ones are described below:

- IsBasicLatin: Latin characters, English, and many European languages fall with this set of code points.
- IsGreek: Greek language code points (range 0370 -03FF).
- IsHebrew: Hebrew language code points (range 0590 -05FF).
- IsCurrencySymbols: Currency Symbol code points (range 20A0-20CF).
- IsMathematicalOperators: Math symbol code points (range 2200-22FF).

Note that the Named Block is case-sensitive, and an error will occur if you use the wrong case or spell the named block wrong. You can find the complete list of the Unicode named blocks supported by the Microsoft .NET regex framework here: Unicode Character Blocks.

Summary

The Unicode standard allows many language systems to be represented by computer systems. Using the .NET regex object, you can search for individual Unicode characters, Unicode character classes, and Unicode writing systems.

Chapter 15 Optimizing Your Regex

A complicated regex pattern that is processing a large amount of text can be quite a resource hog, both in terms of memory and processing time. In this chapter, we can consider a few things to do to speed your expression up. Keep in mind, the more you know about the data you are searching through, the better you'll be able to optimize your regex performance.

Also, keep in mind that regex processing is very fast, and the optimizations will most likely not be noticeable unless you are trying to save milliseconds. If you need to spend a few hours optimizing a regex pattern that is used once a month, it will take a long time savings milliseconds to recoup your development effort. Consider the optimizations as you design your regex, then you don't need to come back and revisit the code.

Alternation

Although the processing difference is miniscule, it makes sense to put the more likely occurrences at the beginning of the alternation list. For example, if you are writing a regex expression to validate domains for websites, you should be listing **com**, then **net**, then **org**, etc. This is a general rule of thumb for any sequential search processing: put the most likely matches at the beginning of the list.

Lazy and Greedy Quantifiers

Lazy quantifiers potentially will run quite a bit faster than greedy quantifiers if the search string is relatively small and close to the beginning of the text. For example, if we use the greedy quantifier <H3>.*</H3> to look for tags in a large HTML document, the process will load the entire HTML document into memory. Trying to find an <H3> pattern in the source code to my Stack Overflow profile took about 8 milliseconds using the greedy * pattern. Using the lazy pattern, the same search ran in less than .5 milliseconds.

Of course, the reverse holds true if you are searching for a pattern that you'd expect to be the entire string or at least most of it. In this case, you'd rather stick with the greedy operator, anticipating a more likely match to occur quicker by looking at the entire string.

Non-Capturing Groups

If you are relying on groups for your processing, you should only capture the groups you plan on using in the expression.

```
(?: <pattern> )
```

The above syntax creates a non-captured group based on the <pattern> indicated. For example, see the following pattern:

```
(?<first>\w+)\s(?:\w+)\s
```

To extract the first two words of a sentence, when you can discard the second word, runs about a millisecond faster than naming the second group.

Compiling Expressions

If you set the Microsoft compile option on, then the regex will first be compiled and stored in the cache. The first time the regex is encountered, the regex engine will compile it and put it into the cache. Subsequent calls to the same regex pattern will run much quicker.

Cache Size

The default cache size is 15 entries for regular expressions. You can set the cache size using the **CacheSize** property. It is a static property on the regex object that can be set to 0 to clear the cache or any number to change the cache size.

Disable Backtracking

Backtracking is a process used by the regex engine to try alternative paths if the first branch fails. If you know that backtracking will not succeed, you can instruct the regex engine not to bother even trying. For example, we need a simple regex pattern to check if a file contains a valid HTML syntax, or if the text begins with **<html>** and ends with **</html>**.

```
(<html>.*</html>)
```

If you use the pattern above, then the regex engine will look at the file and see that the first tag is <html> and the last tag is </html>. If not, the regex engine will try other paths through the text to see if it a match. However, we know that if such a match is not found, then the file is not valid for our purposes.

By adding the ?> non-backtracking characters to our expression, it becomes the following:

```
(?><html>.*</html>)
```

We instruct the engine to give up if a first attempt fails. On a good size text, I see a 3 to 7 millisecond savings by eliminating backtracking. However, for our purposes it will most likely fail.

Note: If you use the .* syntax, remember to include the SingleLine option, since the . will normally not match line-feed characters, which will most likely be found in your text file.

Use Anchors

Another optimization is to use anchors when possible, particularly at the start and end of line or string anchors. This allows the regex engine to simplify its search to check the entire string. If the entire string does not match the pattern, and the anchors characters indicate the pattern must be found at the beginning or end, then the regex engine can quickly “fail” if the pattern expected is not found at the anchor position.

Summary

Optimization of regex processing is a habit you should get into at the start instead of trying to optimize the regex after it has been tested. The better you know your data, the more likely you can apply optimizations to allow the regex to find (or not find) the result more quickly.

Chapter 16 Regex Errors

If you make an error in your regex syntax, the Microsoft engine will trigger an exception. The exception class that is triggered is the **System.ArgumentException** class (used for any method with argument exceptions). Although there is not a lot of extra detail in the exception, the message text is generally pretty descriptive of the regex error.

Catching the Error

There are several type of errors that can occur processing regular expressions. These include:

ArgumentNullException

This exception will occur when the regular expression pattern is null. It is a child exception to the System Argument exception, but you could add it to the exception chain if you wanted to distinguish between null and invalid syntax errors.

Note: A null pattern will trigger the exception, any empty string will not. An empty string is technically a valid regex pattern, although it will not match anything.

RegexMatchTimeoutException

This exception is triggered when the regex engine times out by attempting to process the regular expression pattern. With .NET 4.5, you can specify a timeout on the regex constructor; older versions have infinite time limits.

You can use a standard try catch block to capture any errors in regular expression patterns.

```
try
     {
        string pattern = @"\b((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}\b");
        string source = "(610) 555-1212";
        Regex theExpr = new Regex(pattern);
        Match theMatch = theExpr.Match(source);
        if (theMatch.Success)
           {
             // Do some processing…
           }
      }
 catch (ArgumentNullException ex)
```

```
        {
           // Pattern is null.
        }
   catch (RegexMatchTimeoutException ex)
        {
           // Expression exceed time-out limit (.NET 4.5)
        }
  catch (Exception ex)
        {
          // ex.Message contains descriptive text of the syntax error
        }
```

Working with Groups and Captures

If you are working with the match object (or a group object within a match), one issue to be aware of is that groups and captures are handled differently.

Groups

Imagine we have a simple regular expression to extract the numeric portion of a day from a calendar. The calendar day might be entered as a number, such as 4, or it might be entered with text afterwards, such as 22nd. We are using the following regex pattern:

(?<digit>[0-9]+)(?:st|nd|rd|th)?

We used this regex to extract just the numeric portion. Our code looks like the following:

```
   Regex theExpr = new Regex(pattern);
   Match theMatch = theExpr.Match(source);
   if (theMatch.Success)
      {
         // Do some processing…
         int theDigit = Convert.ToInt16(theMatch.Groups["digit"].Value);
      }
```

This code will work as expected, and will attempt to convert the value of the group into an integer. However, the following code will not raise an error, but also won't give you the expected results:

```
  string theDay = theMatch.Groups["Digit"].Value;
  string theLetters = theMatch.Groups[2].Value;
```

When an invalid group name or number is specified, a group object is returned with the **Success** property set to false and the **Value** property set to an empty string. In the first case, the group name was “digit,” not “Digit” (case-sensitive), so the group is an empty non-match. In the second case, there is no group number two, so the empty group is returned.

Tip: Remember that group names are case-sensitive, and if the group is not found, an empty group is returned rather than an exception triggered.

Captures

The captures collection behaves similarly to most .NET collections. If you attempt to reference an invalid capture element, you’ll trigger an ArgumentOutofRange exception. Also, keep in mind that the **Captures** collection is zero-based, while the **Groups** collection is one-based.

Chapter 17 Other Regex Options

Microsoft's regex object has a few additional options beyond the ones we've covered in this book. Two of these options are particularly useful when working with international text.

CultureInvariant

When the regex engine performs case-insensitive comparisons, the casing rules of the current culture are used to map equivalent uppercase and lowercase letters. If you need to use a particular culture, you can use the following code to switch cultures used for processing the regex pattern:

```
  CultureInfo defaultCulture = Thread.CurrentThread.CurrentCulture;
  Thread.CurrentThread.CurrentCulture = new CultureInfo("tr-FR");

  // Process the regex, using tr-FR (French) casing rules.

  Thread.CurrentThread.CurrentCulture = defaultCulture;  // Restore default
culture.
```

Note: You will need to include the System.Globalization and System.Threading assemblies to use the above code.

For most languages this is acceptable; however, there are some languages where the comparison might not yield the expected results. In particular, the Turkish language has two lowercase "i's," one dotted and one not. When they are converted to uppercase, two uppercase characters are used: one without the dot and one that keeps the dot. If your regex pattern doesn't account for this, you might not get the matches you expected.

CultureInvariant Regex Option

By adding the CultureInvariant regex option, you request that the special Invariant Culture be used for processing the regex. This culture is a special fake culture (based on English) which is available in the System.Globalization name space. This culture handles issues and other cultural variations. In addition to being used for regex processing, it is handy for any data that needs to be stored and retrieved without regard to the culture at the time.

RightToLeft

Another option to consider is that some languages, such as Arabic and Hebrew, are read from right to left. In these languages, you would want to have your regular expression search the same way, starting at the right. Adding the **RightToLeft** regex option will cause the pattern matching to start at the rightmost character and move towards the left.

While this is useful in Arabic and Hebrew languages, you can also use it to find the last item in a list of items. For example, if we had a text file of birthdays and want to get the year associated with the first one, we could use the regex pattern as follows: `[ /]\d{4}.`

```
March 3rd, 1926
July 25th, 1933
June 21st, 1957
July 5th, 1958
9/12/1959
3/3/1961
```

With regular processing, the year 1926 would be returned. By setting the right to left option, we get the year 1961 back instead. You could use this approach if you want to find the last <p> tag added to an HTML document, etc.

Note: The regex object has a Boolean property RightToLeft to determine if the Right to Left option has been set.

ECMAScript

ECMAScript is a scripting language standard that is often used for client-side scripting, most notably in JavaScript. ECMAScript has support for regular expressions, but some of the features differ from the Microsoft regex implementation. You can use the ECMAScript regex option to cause the regex object to follow the ECMAScript implementation. The notable differences are described below.

Regex Options

When the ECMAScript option is enabled, the only other available options are the IgnoreCase and multi-line options. Trying to combine any other options with ECMAScript will result in an Argument Out of Range exception.

Character Classes

ECMAScript does not support Unicode characters, so any string with Unicode characters will not match using the ECMA option. If we have Japanese text, such as こんにちは, the pattern \w+ will match it using a Microsoft regex, but will fail to match if we use the ECMAScript option.

Back References

A back reference is a \ character followed by a numeric value. However, an octal escape sequence can also be a \ followed by a numeric. The regex engine needs to decide how to interpret such a sequence, as either a back reference or an octal escape.

Table 18: Back References

Pattern	Microsoft Regex	ECMAScript
\0dd	As long as dd is a valid octal, treat as octal escape	Same
\[1-9]	Always treat as a back reference	Back reference if group exists, otherwise interpret as literal value
\[1-9]dddd	If group exists, assume back reference; otherwise, interpret up to 3 digits as octal and remaining digits as literal value	Interpret as back reference by using as many digits as match; if no match, assume octal for first 3 digits and literal values beyond that

While the differences are minor between the two implementations, you could see some issues when a valid Microsoft regex pattern works differently in JavaScript. By using this option, you can at least simulate to a degree how JavaScript is likely to handle a particular regex pattern.

Chapter 18 Regex Summary

The following table shows all of the regex options in a quick summary cheat sheet.

Table 19: Character Options

Metacharacters	
^	Start of string (or line if multiline option used)
$	End of string (or line)
.	Any character (except \n – or all characters if single line option used)
|	Alternation
{n,m}	Match specific number of occurrences
[…]	Match one character from list
(….)	Create a group
*****	0 or more of previous pattern
+	1 or more of previous pattern
?	0 or 1 of previous pattern, also converts greedy expressions to lazy
****	Changes following meta character to literal character
\<1-99>	Back reference to a group in the pattern, numbered 1-99
Character Classes	
[aeiou]	Match any single character between brackets
[^aeiou]	Match any single character not in list between brackets
[0-9]	Matches any character within range. To match a hyphen, place it at beginning or end of the character set
\w	Match any word characters [a-zA-Z0-9_]. For .NET, also matches Unicode
\W	Match any non-word characters, i.e. [^a-zA-Z0-9_]
\s	Matches any whitespace [\f\n\r\t\v]
\S	Matches any non-whitespace [^\f\n\r\t\v]
\d	Matches any digit [0-9]
\D	Matches any non-digit [^0-9]

Character Escapes	
\t	Matches tab character
\r	Matches carriage return
\v	Matches vertical tab
\f	Matches form feed
\n	Matches new line
\e	Matches escape character

Table 20: Unicode Options

Unicode	
\u9999	Matches Unicode code point, must be exactly 4 digits after \u
\p{L}	Matches any Unicode “letter”
\P{L}	Matches any non-letter Unicode category
\p{Z}	Matches any whitespace or invisible separator character
\P{Z}	Matches any non-whitespace Unicode character
\p{N}	Matches any Unicode number
\P{N}	Matches any non-number Unicode character
\p{P}	Matches any punctuation character
\P{P}	Matches any non-punctuation Unicode character
\p{C}	Matches control characters and unused code points
\P{C}	Matches non-control characters in Unicode
\p{S}	Matches any symbol character
\P{S}	Matches any non-symbol character in Unicode
Named Unicode blocks	
\p{block}	Found character in named block IsBasicLatin IsGreek IsHebrew IsArabic

	IsCurrencySymbols IsMathematicalOperators See Unicode Character Blocks for more named blocks in .NET
\P{block}	Does not match named block

Chapter 19 Resources

There are a large number of Internet sites which provide regex examples, tutorials, and online testers. A few sites I've found useful are listed below:

Regex Testers

Regex tester websites allow you to provide a regular expression and some search text, and then apply the pattern to see the results. They are very similar to our example C# programs, except you can typically try different regex versions such as JavaScript, C#, PHP, etc., to uncover any subtle differences.

Regex101

The site regex101.com has a regex tester that lets you work with PHP, JavaScript, and Python. It will also "explain" your regex pattern, which is often useful when debugging if your regex pattern doesn't quite work as expected. However, this site does not have a .NET regex pattern test option.

Debuggex

The site debuggex.com has a regex tester with a neat visual display of the regex above the expression window. Although you can use the site without registering, providing your email address gives you access to some additional features, and the site includes a library of common regular expressions.

Regex Hero

The site regexhero.net is an online regular expression tester site specifically for .NET regex patterns. A nice feature is that you can test the Split() and Replace() methods as well. It will also generate .NET code to use your regex pattern.

Regex Libraries

There are many websites available with collections of regex patterns. If you need a particularly common pattern, such as phone numbers or dates, you can probably find good pattern examples on the web. One of my personal favorites is regxlib.com. This site has categories of user-submitted regular expressions and allows you to search for patterns. You can also test expressions and contribute your own patterns. There are currently almost 4,000 expressions in the library.

Regex Tutorials

Many of the tester sites also offer tutorials to explain regex processing. One of my favorite tutorial sites is regular-expressions.info. Although the site contains advertisements, they are not too intrusive and the content is worth the visit.

Another good tutorial site is rexegg.com which contains a tutorial, many examples, and tricks of the regex trade. The site has a section to learn the fundamentals and another section for more advanced tutorials. It is definitely worth the visit.

My Personal Favorite

My personal favorite site for us computer types is Stack Overflow. You can ask questions and anyone can jump in and give you answers. If you find yourself stuck with a regex or any topic, visit the site and ask away. Be sure to explain your problem clearly and show what you've tried so far. Or just poke around and see what other issues people have run into. It is a lot more fun than building a farm or castle.

Summary

Regex processing is a powerful tool to add to your toolkit. It allows you to do all sorts of text searching and manipulation. Although the syntax can be a bit terse and hard to follow sometimes, the power is a real-time savings over writing your own text manipulation code. Take the time to experiment and gradually introduce regex patterns into your string code. You'll find it is a nice addition to your programming capabilities.