RegexBuddy

Manual

Version 2.2 — 12 January 2006

Table of Contents

RegexBuddy Manual..... 1

1. Introducing RegexBuddy..... 3
2. Define a Match, Replace or Split Action..... 5
3. Insert a Token into The Regular Expression..... 7
4. Analyze and Edit a Regular Expression..... 10
5. Export The Analysis of a Regular Expression..... 11
6. Testing Regular Expression Actions..... 12
7. Debugging Regular Expressions..... 14
8. Comparing the Efficiency of Regular Expressions..... 16
9. Generate Source Code to Use a Regular Expression..... 19
10. Copying and Pasting Regular Expressions..... 23
11. Storing Regular Expressions in Libraries..... 25
12. Parameterizing a Regular Expression Stored in a Library..... 27
13. Using a Regular Expression from a Library..... 28
14. GREP: Search and Replace through Files and Folders..... 29
15. Adjust RegexBuddy to Your Preferences..... 32
16. Integrate RegexBuddy with Searching, Editing and Coding Tools..... 34
17. Basic Integration with RegexBuddy..... 36
18. Integrating RegexBuddy using Standard Input and Output..... 38
19. Integrating RegexBuddy Using COM Automation..... 41
20. Contact RegexBuddy's Developer and Publisher..... 45

Regular Expression Tutorial..... 47

1. Regular Expression Tutorial..... 49
2. Literal Characters..... 51
3. First Look at How a Regex Engine Works Internally..... 53
4. Character Classes or Character Sets..... 55
5. The Dot Matches (Almost) Any Character..... 58
6. Start of String and End of String Anchors..... 60
7. Word Boundaries..... 63
8. Alternation with The Vertical Bar or Pipe Symbol..... 65
9. Optional Items..... 66
10. Repetition with Star and Plus..... 67
11. Use Round Brackets for Grouping..... 70
12. Named Capturing Groups..... 74
13. Unicode Regular Expressions..... 76
14. Regex Matching Modes..... 80
15. Atomic Grouping and Possessive Quantifiers..... 81
16. Lookahead and Lookbehind Zero-Width Assertions..... 86
17. Testing The Same Part of The String for More Than One Requirement..... 90
18. Continuing at The End of The Previous Match..... 92
19. If-Then-Else Conditionals in Regular Expressions..... 94
20. Adding Comments to Regular Expressions..... 96

Regular Expression Examples 97

1. Sample Regular Expressions 99
2. Matching Floating Point Numbers with a Regular Expression 101
3. How to Find or Validate an Email Address 102
4. Matching a Valid Date 106
5. Matching Whole Lines of Text 108
6. Deleting Duplicate Lines From a File 109
8. Find Two Words Near Each Other 110

Tools & Languages 111

1. What Is grep? 113
2. PowerGREP: Taking grep Beyond The Command Line 115
3. EditPad Pro: Convenient Text Editor for Windows and Linux 117
4. Using Regular Expressions with Delphi for .NET and Win32 119
5. Using Regular Expressions in Java 120
6. Java Demo Application using Regular Expressions 123
7. Using Regular Expressions with JavaScript and ECMAScript 130
8. Using Regular Expressions with The Microsoft .NET Framework 132
9. C# Demo Application 135
10. The PCRE Open Source Regex Library 142
11. Perl's Rich Support for Regular Expressions 143
12. PHP Provides Two Sets of Regular Expression Functions 145
13. Python's re Module 147
14. Using Regular Expressions with Ruby 150
15. VBScript's Regular Expression Support 152
16. VBScript RegExp Example: Regular Expression Tester 155
17. How to Use Regular Expressions in Visual Basic 157

Regular Expression Reference 159

1. Basic Syntax Reference 161
2. Advanced Syntax Reference 165
3. Syntax Reference for Specific Regex Flavors 168
4. Unicode Syntax Reference 169

Part 1

RegexBuddy Manual

1. Introducing RegexBuddy

RegexBuddy is your perfect companion for working with regular expressions. Let me give you a short overview of some of the most important things you can achieve with RegexBuddy.

You can learn all there is to know about regular expressions today with RegexBuddy's detailed, step by step regular expression tutorial. You can find this tutorial in the second part of this manual.

Learn each of the different elements that compose a regular expression, step by step in logical order. If you already have some experience with regular expressions, this logical separation enables you to brush up your knowledge on specific areas. When trying to understand a regex, you only need to click the Explain Token button, and RegexBuddy will present you the appropriate topic in the tutorial.

Regex Tree and Regex Building Blocks

RegexBuddy's regex building blocks make it much easier to define regular expressions. Instead of typing in regex tokens directly, you can just pick what you want from a descriptive menu. Use RegexBuddy's neatly organized tree of regex tokens to keep track of the pattern you have built so far.

When you need to edit a regular expression written by somebody else, or if you are just curious to understand or study a regex you encountered, copy and paste it into RegexBuddy. RegexBuddy's regex tree will give you a clear analysis of the regular expression. Click on the regular expression, or on the regex tree, to highlight corresponding parts. Collapse part of the tree to get a good overview of complex regular expressions.

You can create and edit regular expressions quickly and easily with RegexBuddy. You can mix manipulating RegexBuddy's building blocks and directly editing the regex pattern to suit your own skill and style. Rely on RegexBuddy as you rely on a buddy or coach to assist you.

Regex Tester and Debugger

You should not risk actual data with untested regexes. Copy and paste sample data into RegexBuddy, or open test files. You can step through the search matches in the sample data, and get a detailed report about each match. Or highlight all matches to debug the regex in real time as you edit it.

When you plan to use a regex in a search-and-replace operation, preview the search and replace in RegexBuddy. If you want to split a string using a regex, check the result in RegexBuddy. Avoid nasty surprises when using a regular expression to modify real data or files.

If a regex isn't working exactly the way you'd expect it to, invoke RegexBuddy's debugger to see exactly how the regular expression is applied. You will see which text is matched by each token, at every step during the matching process. You will know exactly why the regex works the way it does, and fix it without any guesswork.

Develop Efficient Software Quickly with Instant Code Snippets

You can save time and code efficiently by using regular expressions when developing applications and scripts. With the proper regex, you can often do in a single line of code, or a few lines of code, what would otherwise require dozens or hundreds.

Rely on RegexBuddy to handle the details, such as which classes and function calls to use, and how to escape special characters. Just select the language you are working with, and the action you want to perform. Test whether a string matches a regex, extract matches from a string, search and replace, split a string, etc. RegexBuddy knows all the common regex actions and how to perform them with a variety of programming languages: C#, VB.NET, Java, Perl, PHP, JavaScript and C/C++.

Your Own RegexBuddy Library

Build your own collection of handy regex patterns, and use them whenever you want to. You can easily browse through and instantly search through the regexes you collected. When you found the regex you want, click the Use button.

For common tasks, use one of the many regular expressions you can find in RegexBuddy's library of pre-created regular expressions. You will find readily useful regexes for a wide variety of tasks. For many tasks, there will be several choices of regex patterns, with the differences clearly described.

2. Define a Match, Replace or Split Action

With most tools and languages, you can perform three actions using a regular expression: match, replace or split. You can define, test, debug and implement all three with RegexBuddy.

Match

A match action applies the regular expression pattern to a subject string, trying to find a match. If successful, the match can be applied again to the remainder of the subject string, to find subsequent matches. When you perform a search using a regular expression in a text editor, you are technically executing a match action. The file you are editing is the subject string.

When programming, you can use match actions to validate user input and external data. E.g. the regex «`\A-?\d\Z`» checks if an integer number was entered. Match actions make it easy to parse and process data. Use capturing groups to extract just the data you want.

To define a match action in RegexBuddy, click on the Match tab near the top of the RegexBuddy window. Enter the regular expression into the text box. You can right-click the text box to insert regex tokens without having to remember their exact syntax.

Replace

A replace action applies the regular expression pattern to a subject string, and replaces the match or each of the matches with a replacement string. Depending on the tool or language used, this will either modify the original subject string, or create a new string with the replacements applied. By using backreferences in the replacement text, you can easily perform some pretty complex substitutions. E.g. to invert a list of assignments, turning "`one = another`" into "`another = one`", use the regex «`(\w+)\s*=\s*(\w+)`» and replace with "`$2 = $1`".

To define a replace action in RegexBuddy, click on the Replace tab. Enter the regular expression in the topmost text box, and the replacement text into the text box just below. To easily insert a backreference into the replacement text, first left-click in the text box with the regex, to move the text cursor inside the capturing group you want to insert. Then, left-click in the text box with the replacement text to place the text cursor at the spot where you want to insert the backreference. Finally, right-click the replacement text box and select "use backreference as is". A backslash followed by the number identifying the capturing group will be inserted.

Many programming languages use dollar signs rather than backslashes to insert capturing groups into the replacement text. RegexBuddy will automatically convert the backslashes into dollar signs when you implement the replace action on the Use tab. If you enter the backreference manually into the replacement text, you can use dollar signs. RegexBuddy supports both styles.

EditPad Pro and PowerGREP have the ability to control the case of the inserted backreference. `\U2` inserts the second backreference in upper case, `\L2` in lower case, `\i2` with initial caps, and `\F2` with the first character in upper case, and the remainder in lower case. RegexBuddy supports this too. However, when you implement the replace action on the Use tab, the case conversion option will be lost. None of the programming languages RegexBuddy supports offer a similar facility.

Split

A split action creates an array or list of strings from a given subject string. The first string in the resulting array is the part of the subject before the first regex match. The second string is the part between the first and second matches, the third string the part between the second and third matches, etc. The final string is the remainder of the subject after the last regex match. The text actually matched by the regex is discarded.

You can specify a limit for the split action. The limit is the maximum number of strings the resulting string or array will contain. In other words, the string is split at most limit-1 times. The last string in the result is the unsplit remainder of the subject. If the regex can be matched fewer than limit-1 times, the resulting array will have fewer items than your limit.

If you are interested in the 3rd item of a comma-delimited list, you could split with the trivial regex «,» with a limit of 4. The third item in the array is then the 3rd comma-delimited item in the subject.

In RegexBuddy, simply click on the Split tab and enter the regular expression into the text box. You can right-click the text box to insert regex tokens without having to remember their exact syntax. You can specify the limit in the spinner box below the text box with the regular expression. If it is invisible, make the RegexBuddy window wider. A limit of zero or one means there is no limit at all.

3. Insert a Token into The Regular Expression

RegexBuddy makes it easy to build regular expressions without having to remember every detail of the complex regular expression syntax.

To insert a token when defining a regex action, first left-click in the text box to move the text cursor to the spot where you want to insert the token. Then right-click to access the Insert Token menu.

Some tokens, including lookaround, are grouping tokens. You can insert a group on its own. RegexBuddy will then put the text cursor inside the newly added group, so you can fill it right away. To place an existing part of the regular expression inside the new group, first select that part, and then add the grouping token.

Quantifiers are a special case. If you select part of the regular expression before inserting a quantifier, RegexBuddy will turn the selection into a non-capturing group and apply the quantifier to that. If not, the quantifier is inserted as a normal token, and it will repeat whatever precedes it.

When analyzing a regular expression on the Create tab, you can easily designate the spot where you want to insert the token. Click on a token in the regex tree, and the new token will be inserted right after it. Click on the Insert Token button in the toolbar to access the Insert Token menu.

List of Regex Tokens

Literal text: Enter one or more characters that will be matched literally. RegexBuddy will escape any metacharacters you enter.

Non-printable character: Match a specific non-printable character, such as a tab, line feed or carriage return. You can insert non-printable characters directly into the regular expression, or inside a character class.

Dot: Matches any single character, except newline characters (unless you turned on “dot matches newline”)

ASCII Character: Matches a specific character from the current ASCII or ANSI code page. RegexBuddy will show you a grid with all available characters. Above the grid, first choose if you want to match only one particular character, or if you want to match one character from a number of possible characters. If you select to match one character, click on the character in the grid and then click OK. Otherwise, clicking on a character in the grid will toggle its selection state. Select the characters you want, and click OK.

Unicode Character: Matches a specific Unicode character. Select one character or multiple characters in a grid as explained for inserting an ASCII character. Since the Unicode character set is very large, you can make the grid show only characters of a certain category to make it easier to pick the character you want. If you see a great number of squares instead of characters in the grid, click the Select Font button to change the grid's font. The squares indicate the font cannot display the character.

Short-hand character class: Matches a single character from a specific set of characters. You can insert short-hand character classes directly into the regular expression, or inside a character class.

Character class: Matches a single character out of a set of characters. You can specify a list of individual characters and/or one or several ranges of characters. A range matches any character with an ASCII value between the first and the last character in the range. Though you can use any pair of characters to define a

range, it is strongly recommended to use only letters a..z and digits 0..9 for ranges. RegexBuddy will escape any characters that are metacharacters inside the character class.

Anchors: Anchors match a position, rather than characters:

- Start of string: «\A» matches at the very start of the string only.
- Start of string or after a line break: «^» matches at the very start of the string. Also matches after each line break in the string if the option "^$ match at line breaks" is checked.
- End of string: «\z» matches at the very end of the string only.
- End of string or before terminating line break: «\Z» matches at the very end of the string, or before the last character in the string if that is a newline character.
- End of string or before a line break: «$» matches at the very end of the string, or before the last character in the string if that is a newline character. Also matches before every other line break in the string if the option "^$ match at line breaks" is checked.
- Word boundary: «\b» matches between a word character and a non-word character, as well as between a word character and the start and the end of the string.
- Non a word boundary: «\B» matches between two word characters, as well as between two non-word characters.
- End of previous match: «\G» matches at the end of the previous match, or at the very start of the string during the first match attempt.

Alternation: Alternation causes the overall regex or group (if inserted inside a group) to match if either the part to the left of the vertical bar, or the part to the right of the vertical bar can be matched. Insert multiple vertical bars to create more than two alternatives.

Quantifier: Specify the minimum number of times the preceding token must be matched, and the maximum number of times it may be matched. If the maximum is greater than the minimum, specify if the quantifier should be greedy (trying the highest possible number of times first, then descreasing), lazy (trying the smallest possible number of times first, then increasing), or possessive (trying the highest possible number of times only).

Capturing group: Group the selected tokens together and store their part of the match in a numbered backreference. Numbered capturing groups are automatically numbered from left to right, starting with number one.

Named Capturing group: Group the selected tokens together and store their part of the match in a named backreference. RegexBuddy will ask you for the name of the group, and the regex flavor you are working with. Both the .NET languages and Python support named capture, using incompatible syntax. RegexBuddy supports both variants. RegexBuddy will automatically convert between the .NET and Python variants when generating code snippets, but not when you copy and paste the regular expression.

Non-capturing group: Group the selected tokens together, so you can insert a quantifier behind it that will apply to the entire group. Does not create a backreference.

Use backreference: In the window that appears, click inside the capturing group or named group to which you want to insert a backreference. The backreference will match the same text as last matched by the capturing group. If you point at a named group, the backreference will use the same syntax (.NET or Python) as you used for the group.

Atomic group: Create a capturing group that is atomic or indivisible. Once an atomic group has matched, the regex engine will not try different permutations of it at the same starting position in the subject string.

Lookaround: Matches at the current position if the test succeeds. Does not match at all if the test fails.

- Positive lookahead: Succeeds if the regular expression inside the lookahead can be matched starting at the current position.
- Negative lookahead: Succeeds if the regular expression inside the lookahead can NOT be matched starting at the current position.
- Positive lookbehind: Succeeds if the regular expression can be matched ending at the current position (i.e. to the left of it)
- Negative lookbehind: Succeeds if the regular expression can NOT be matched ending at the current position (i.e. to the left of it)

Comment: Add descriptive text to the regex. Comments are ignored by the regex engine.

4. Analyze and Edit a Regular Expression

RegexBuddy takes away all the obscurity and confusion caused by the rather cryptic regular expression syntax when you click on the Create tab.

When you enter a regular expression or paste in an existing regex, the Create tab shows you a tree outline of the regular expression, called the “regex tree”. The regex tree describes the regular expression in plain English, using a structurethat you can easily navigate.

Click on a node in the regex tree, and RegexBuddy will highlight the corresponding token in the regular expression. While you edit the regular expression, RegexBuddy keeps the regex tree synchronized with the updated regular expression. As you move the text cursor while editing, RegexBuddy will highlight the node in the tree that describes the token around or immediately to the left of the text cursor.

To get more information about a particular node, click on the node and then click the Explain Token button. This will open the regex tutorial in RegexBuddy's help file, right at the page that describes the node you selected.

After clicking on a node, you can press the Delete key on the keyboard to delete it, or the Insert key to insert a regex token. You can rearrange parts of the regular expression by dragging and dropping nodes with the mouse. Note that moving a node this way actually moves the part of the regular expression that the node represents, rather than the node itself. The regex tree is rebuilt after you moved the node. This can have side effects. E.g. if you move a quantifier to the very start of the regular expression, the node will turn from a quantifier node into an error node.

RegexBuddy gives you full freedom to edit the regular expression. It does not get into your way by preventing the regular expression from becoming invalid. Instead, RegexBuddy assumes you know what you are doing, and helps you with that by clearly analyzing your regular expression, whatever state it is in.

The regex tree is not only handy to analyze a regular expression while editing it. The regex tree is perfect for explaining to others how a regular expression works. You can include the regex tree in your documentation or teaching materials by clicking the Export button.

5. Export The Analysis of a Regular Expression

The regex tree RegexBuddy shows on the Create tab is perfect for explaining to others how a regular expression works. You can include the regex tree in your documentation or teaching materials by clicking the Export button. A small menu will pop up below the button, giving you the choice between exporting to a plain text file, and exporting to an HTML file.

Export to a Plain Text File

When you export the regex tree to a plain text file, RegexBuddy will ask you for the name of the file, and for a caption. The caption is placed at the top of the text file, followed by the regular expression and then the regex tree. Each node in the tree will occupy one line in the text file, indented with spaces. When viewing the text file in a text editor, turn off word wrap to clearly see the tree structure.

Export to an HTML File

When you export the regex tree to an HTML file, RegexBuddy will ask you for the name of the file, and for a caption. The caption is used for the title and header tags in the HTML file. You can also choose if you want the HTML file to link to http://www.regular-expressions.info or not. If you do, each node in the regex tree will link to the page in the tutorial on this web site that explains the node. If you plain to upload the HTML file to your web site, you should definitely include the links. This will makes it much easier for your site's visitors to learn more about regular expressions. The tutorial at http://www.regular-expressions.info is the same tutorial as the one in RegexBuddy's documentation. It was written by Jan Goyvaerts, who also designed and developed RegexBuddy.

When you view the HTML file exported by RegexBuddy, you will see the regular expression at the top of the page, and the regex tree as a bulleted list below it. If your web browser supports style sheets and JavaScript, moving the mouse pointer over the regular expression or the regex tree causes part of the regular expression and the corresponding node in the regex tree to be highlighted. This makes it very convenient to grasp which part of the regex does what, just like clicking on regex tree nodes does on the Create tab in RegexBuddy.

6. Testing Regular Expression Actions

It is always a good idea to test your regular expression on sample data before using it on valuable, actual data, or before adding the regex to the source code for the application you are developing. You can debug a regular expression much easier with RegexBuddy.

It is obvious that you should check that the regex matches what it should match. However, it is far more important that you verify that it does *not* match anything that you do *not* want it to match. Testing for false positives is far more important, something which people new to regular expressions often do not realize.

An example will make this clear. Suppose we want to grab floating point numbers in the form of „`123`", „`123.456`" and „`.456`". At first sight, the regex «`\d*\.?\d*`» appears to do the trick.

To test this, type "`The number is 14.95`" in the edit box just below the toolbar on the Test tab. Then click the Find First button. RegexBuddy reports a zero-length match at offset 0. If you click Find Next, RegexBuddy reports a zero-length match at offset 1. Another click on Find Next gives a match at offset 2, and so on until „`14.95`" is matched at offset 14.

While our regular expression successfully matches the floating point numbers we want, it also matches an empty string. Everything in the regex is optional. The solution is to make the regex enforce the requirement that a floating point number must contain at least one digit. See the floating point number example for a detailed discussion of this problem.

Preparing Sample Data to Test Your Regex

The Test tab has two edit boxes. The top one is where you enter the subject string to test the regex on. You can simply type in some text, or click the Open File button to use a text file as the subject string. You can access recently used test files by clicking the downward pointing arrow of the Open File button.

If you copied some text to the Windows clipboard, you can use that as the test subject by clicking the Paste Subject button. Paste "as is" to paste the text on the clipboard unchanged. If you copied a string literal from your source code, including the single or double quotes delimiting it, select one of the programming language string options when pasting. Select C-style for C++, Java, C#, etc., Pascal-style for Delphi, Basic-style for VB, etc.

Performing The Tests

Click the Debug button to invoke the RegexBuddy debugger at the position of the text cursor in the test subject. RegexBuddy will automatically switch to the Debug tab.

Click the Highlight button to highlight all regex matches in the test subject. The matches will be highlighted with alternating colors. That makes it easier to differentiate between adjacent matches. If you put capturing groups into the regex, you can highlight the part of each match captured into a particular backreference. That way you can test whether you have the right number for the backreference, and whether it properly captures what you want to extract from each match. Click on the downward pointing arrow on the Highlight button to select the number of the backreference to highlight. The overall matches will still be highlighted. The

highlighting for the backreference will alternate between two colors, along with the overall match. You can change the colors in the preferences.

When matches are highlighted, click the Previous button to jump to the match before the current position of the text cursor. Click the Next button to jump to the first match after the cursor. The Find First and Find Next buttons are not available when matches are being highlighted. To get more information about a particular match, double-click it. The bottommost text box will show the text matched and its offset and length, for the overall match and all capturing groups. The number of matchesis also indicated.

The Find First button applies the regular expression once to the entire subject string. The matched part of the string is selected in the edit box, and the text cursor is moved to the end of the match. Details about the overall match, and details about each capturing group, are shown in the text box at the bottom. If the regular expression cannot match the subject string at all, the bottommost text box will tell you so.

The Find Next button works just like the Find First button, except that it applies the regex only to the part of the subject string after (i.e. to the right and below) the text cursor's position in the subject edit box. If you do not move the text cursor yourself between clicking on the Find First and/or Find Next buttons, the effect is that Find Next continues after the previous match. This is just like a regular expression engine would do when you call its “find next” function without resetting the starting position, which is also what the Highlight and List All buttons do.

If you want to test the match from a particular starting position, just click at that position to move the text cursor there. Then click Find Next. This can lead to different match results than indicated by the Highlight and List All buttons.

When defining a match action, the List All button will be visible. When you click this button, RegexBuddy will perform the search through the entire subject string. The list of matches is shown in the text box at the bottom. Double-click a match in the list to select it in the subject text.

When defining a replace action, the Replace All button will be visible. When you click this button, RegexBuddy will perform the search-and-replace across the entire subject string. The result is shown in the text box at the bottom. Replacements will be highlighted in the result. Double-click a replacement to select the text that was replaced in the test subject.

When defining a split action, the Split button is visible instead. RegexBuddy will then split the subject string, and show the resulting list of strings in the edit box at the bottom. The strings in the list are separated by horizontal lines. If the subject string consisted of multiple lines, it is possible that some of the strings in the resulting list span more than one line.

7. Debugging Regular Expressions

RegexBuddy's regular expression debugger provides a unique view inside a regular expression engine. To invoke the debugger, switch to the Test tab. Place the text cursor in the test subject at the position you want to debug the regex match attempt. Then click the Debug button. If regex matches are highlighted in the test subject, placing the cursor in the middle of a highlighted match will debug the match attempt that yielded the match.

After you click the Debug button, the Debug tab will display each step in the match attempt. Each step is the result of one of two possible events. Either a token was successfully matched. The step will indicate the text matched so far by the overall regular expression, including the match added by the last token. The engine then continues with the next token.

The other event is a token that failed to match. The engine will then backtrack to token preceding the one that failed. The step will indicate the text matched thus far, followed by “backtrack”. Backtracking means to go back in the regular expression, not necessarily in the text being matched. In case of greedy or possessive repetition, backtracking forces to preceding token to give up part or all of its match. In case of lazy repetition, backtracking makes the preceding token expand its match. If the efficiency of a regular expression is important, you should try to minimize either kind of backtracking.

How to Inspect the Debugger's Output

To see which text was matched by a particular token at each step, click on the token in the regular expression. If the token consists of a single character, place the text cursor to the right of the token. Alternatively, you can switch to the Create tab, click on the token in the regex tree, and then switch back to the Debug tab.

The text matched by that token is then highlighted in yellow in the debugger's output. You can change the colors in the preferences. If the token is inside a group, its matches will only be highlighted in the steps where the regex engine is processing the tokens inside the group. When the regex engine leaves the group, the text matched by the tokens inside the group is considered to have been matched by the group rather than the tokens inside it. If you place the text cursor after the closing round bracket of a group, the situation is reversed. This way you can see the dynamics between the tokens in the group, and the group itself.

To find out which regex token matches a particular piece of text or character, simply double-click on it in the debugging output. The token will then become selected in the regular expression. This will also cause all the text matched by that token to be highlighted in yellow. If you then switch to the Create tab, the token's regex block in the regex tree will also be selected.

Differences Between The Debugger and a Real Regex Engine

The good news is that RegexBuddy's regular expression debugger is an extension of a real regular expression engine. You will get exactly the same (final) results on the Debug tab as you do on the Test tab.

However, to make it easier to follow the debugger's output, most optimizations the regex engine makes are disabled in the debugger. This is done to maintain a one-on-one relationship between the matching process and the regular expression you typed in. These optimizations do not change the final result, but often reduce the number of steps the engine needs to achieve the result.

The most significant difference occurs with a regular expression such as «`(?:abcd|abef)`». Normally, RegexBuddy's regex engine, like most other regex engines, optimizes this internally into «`ab(?:cd|ef)`». With long alternation sequences, this can reduce the amount of needless backtracking quite a bit.

An optimization that the debugger does make is to keep an initial lookbehind for last. If you debug the regex «`(?<=before)after`», the engine first attempts to match “after”, and will test the “before” lookbehind only if “after” can be matched. This saves the engine from having to search backwards at each character position during a search.

8. Comparing the Efficiency of Regular Expressions

The efficiencyof a regular expression depends on a lot of factors. Each regular expression engine has different optimizations. The text you're applying the regular expression to also has a great impact. The more positions in the text where the regular expression can be partially matched, the more time the regex engine will spend on those failed attempts. All this means that there's straightforward way to benchmark regular expressions using the traditional (virtual) stopwatch.

Still, RegexBuddy's debugger can give you a good view of the overall complexity of a regular expression. Essentially, the more steps the debugger needs to either find the match or declare failure, the more complex the regular expression. Particularly steps marked as “backtrack” are expensive.

When comparing the efficiency of regular expressions, you should run the debugger both on highlighted matches, as well as at positions where the regular expression cannot be matched. E.g. when comparing regular expressions to match a particular HTML tag, place the text cursor on the Test tab before the < of an HTML tag that should not be matched. Click the Debug button to see how many steps it takes to figure out that tag shouldn't match.

Sometimes, the performance difference between two regular expressions can be quite severe. In the regular expression tutorial, there's an example illustrating what is called catastrophic backtracking. The faulty regular expression needs 48,066 steps to fail a short string, a number that grows exponentially with the string's length. The improved regular expression needs only a constant 26 steps.

When comparing two regular expressions, first observe how the number of steps RegexBuddy needs grows. If one regex uses a constant number of steps while the other needs more steps for longer strings, the constant regex is by far the best, even if it needs more steps. However, don't grid your teeth trying to make it constant. It's often impossible for complex text patterns.

If neither regex is constant, compare their growth rate relative to the length of the string. You'll need to test at least three string lengths. First, check if the growth is linear (i.e. the number of steps roughly doubles when you double the length of the string). If one regex grows linearly and the other grows exponentially, the linear regex is always better, even if it needs more steps. As the length of the string grows, the exponential regex will soon exhaust the capabilities of any regular expression engine.

Comparing the actual number of steps only makes sense if both regular expressions are either constant or linear. The one with fewer steps wins. If both are exponential, start with trying to make one of them linear. It's not always possible, but certainly worth trying.

Benchmark Both Success and Failure

You should always benchmark your regular expression both on test subjects that match, and test subjects that don't. In fact, the performance killer is usually slow failure rather than slow matching. Often, a regular expression is used to extract small bits from a larger file. In that cause, the regular expression's performance at positions in the file where it cannot match is far more important than its performance at matching. If you want to extract 10 strings of 100 characters from a file that's a million bytes large, the regular expression will have to match 10 times, and fail 999,000 times.

Writing a regular expression that matches linearly is much easier than writing one that fails linearly. When a regex fails, the regular expression engine will not give up until it's tried all possible permutations of the quantifiers in your regular expression. You'll be surprised how numerous those are.

Making Regular Expressions More Efficient

There are two important techniques to make an exponential regex linear. The easiest and most important one is to make adjacent regex tokens mutually exclusive whenever possible. When writing a regex that locates delimited content, and the delimiters cannot appear (in escaped form) in the content, specify that in your regular expression. Doing so makes sure that the regex engine will not attempt to include the delimiter as part of the content, which significanly reduces the number of pointless permutations the regex engine will try.

As a test, compare the regular expressions «`"[^"]*"`» and «`".*?"`» on the test subject „`“test”`”. Both regexes match, but the former needs 4 steps in the debugger, while the latter needs 11. Now test „`“this is a test”`”. The former still needs 4 steps, but the latter needs 31. Clearly, a greedy negated character class is more efficient than a lazy dot. In actual search time, the impact of this is won't be measurable when your input files only contain short strings. But if you're searching thousands of files with strings thousands of characters long, it will be.

The second technique is the use of atomic grouping and/or possessive quantifiers. Both these features are fairly recent addition to the regular expression culture. RegexBuddy supports both, but your programming language might not. These regex tokens come in handy when you can't use negated character classes. Basically, an atomic group locks in the part of the text the regex matched so far. It says: when you've reached this point, don't bother going back to try more permutations. If you can't find a match, fail right away.

In many situations, you cannot replace a lazy dot with a negated character class to prevent the delimiter from being included in the content. Then you can use an atomic group to achieve the same result. Simply place the atomic group around the lazy dot (or whatever repeated regex token that shouldn't match the following delimiter) and the delimiter that follows it. Then, as soon as the delimiter is matched, the atomic group is locked down. If the remainder of the regex fails, the lazy dot won't get the chance to expand itself and gobble up the delimiter.

See The Difference in RegexBuddy

The tutorial topic on atomic grouping includes a detailed example. You can easily see the effect in RegexBuddy. The two regular expressions are included in RegexBuddy's library as “HTML file” and "HTML file (atomic)". Copy and paste the following example into the Test page:

```
<html>
<head>
   <title>This is a test</title>
</head>
<body>
<h1>This is a test</h1>
<p>First paragraph</p>
<p>Second paragraph</p>
</body>
</html>
```

Both regexes will highlight the whole test subject. If not, turn on the option “dot matches newline” (happens automatically if you use the regexes from the library). If you click on the highlighted match and then click the Debug button, RegexBuddy will find the match in 272 steps for the first regex, and 278 steps for the atomic regex.

Now, delete the very last > character from the test subject. Move the text cursor immediately to the left of the very first < character in the file. Then click the Debug button. The first regex needs 1597 steps to conclude failure. The regex using atomic grouping needs only 291 steps.

If you look closely at the debugger output, you'll see that the first regex produces some kind of vertical sawtooth, extending all the way, backing up a little, extending all the way, backing up some more, etc. The second regex, however, gradually matches the whole file, and then drops everything in just one step.

As a final test, type “`1234567890`” at the end of the test subject, so it ends with “`</html1234567890`”. Put the text cursor back before the very first < character in the file, and click Debug. The regex without atomic grouping now needs 1737 steps, while the atomic regex needs only 311. The 10 additional characters add 140 steps (14-fold the number of characters we added) for the first regex, but only 20 (2-fold) for the second.

By using atomic grouping, we achieved a 7-fold reduction in the regular expression's complexity. Both regular expressions are in fact linear, but two steps per character is a huge savings from fourteen steps per character. If you wonder where these numbers come from, the regex has seven lazy dots. The lazy dot will match a character (one step), proceed with the next token, and then backtrack when the next token fails (second step).

When using atomic grouping, each lazy dot will match each character in the file only once. None of the dots can match beyond their delimiting HTML tag. This yields two steps per character. In the original regex, the dots can backtrack to include their delimiters and match everything up to the end of the file. Since there are seven lazy dots, all characters at the end of the file will be matched by all seven, taking fourteen matching steps.

Had you only compared the performance of these regexes on a valid HTML file, you might have been fooled into thinking the original is a fraction faster. In reality, their performance at matching is the same, since neither regex will do any more backtracking than needed. It's only upon failure that backtracking gets out of hand with the first regex.

9. Generate Source Code to Use a Regular Expression

Once you have defined and tested a regex action, you are ready to use it. One way is to simply tell RegexBuddy to copy the regular expression to the clipboard, so you can paste it into tool or editor where you want to use the regex. If you do this often, you may want to see if it is possible to integrate RegexBuddy and the software you use regular expressions with.

If you want to use the regular expression in the source code for an application you are developing, click on the Use tab in RegexBuddy. RegexBuddy can generate code snippets in a variety of programming languages that have particularly strong support for regular expressions. The examples in C (which has no regex support) illustrate how to use the popular PCRE open source regex library.

Benefits of RegexBuddy's Code Snippets

Using RegexBuddy's code snippets gives you several major benefits. You don't have to remember all the details how to use a particular language's or library's regex support. Though most languages and libraries have similar capabilities, the actual implementation is quite different. With RegexBuddy, you simply select the task you want to accomplish from the list, and RegexBuddy generates source code you can readily copy and paste into your code editor or IDE.

Another key benefit is that when using a regular expression in a programming language, certain characters need special treatment, up and above the special treatment they may need in the regular expression itself. E.g. the regex «\\» to match a single backslash is automatically inserted as “"\\\\"” in a Java or C# code snippet, and as “/\\/” in a JavaScript or PHP/preg snippet. Never again mess around with pesky backslashes!

For replace actions, RegexBuddy also knows if a particular regex library uses backslashes or dollar signs to insert backreferences into the replacement text. RegexBuddy supports both, but most libraries accept only one style.

For languages that support exception handling, RegexBuddy's code snippets also contain exception handling..exception handling or equivalent code blocks to handle any exception that could be thrown by any of the methods the code snippet calls.

How to Generate a Code Snippet

To generate a code snippet, first select your programming language from the list on the Use tab. Then select the type of function you want to implement. Some functions are only available for certain languages. E.g. the functions that create an object for repeatedly applying a regular expression are only available in languages with object-oriented regex libraries.

The available functions also depend on the kind of action you defined. Search and extraction functions are listed for match actions, search-and-replace functions are listed for replace actions, and splitting functions are listed for split actions.

Most functions allow you to specify certain parameters, such as the variable or string constant you want to use as the subject for the regex operation, the variable to store the results in, the name of the regex or

matcher object, etc. RegexBuddy does not verify if you entered a valid constant or variable. Whatever you enter is inserted into the code snippet unchanged. The parameters you entered are automatically remembered. For each parameter, the same values are carried over across all the functions, for each language. So you only need to enter the names you typically use once.

RegexBuddy keeps the code snippet in sync with your regular expression at all times, even when the regex is syntactically incorrect. The snippet is ready for copying and pasting at all times. If you want, you can edit the snippet in RegexBuddy. But remember that your changes will be lost as soon as you edit the regular expression.

You can transfer the snippet into your code editor or IDE by clicking the Copy button on the Use tab's toolbar. This copies the entire snippet to the Windows clipboard, ready for pasting. If you only want to use part of the snippet, either select it and press Ctrl+C on the keyboard to copy it, or select it and then drag-and-drop it with the mouse into your code editor.

Available Languages

- C#, using the System.Text.RegularExpressions library of the .NET framework.
- Delphi (.NET), using the System.Text.RegularExpressions library of the .NET framework.
- Delphi (Win32), using the free TPerlRegEx VCL component.
- Java, using the java.util.regex package part of the JDK since version 1.4.0.
- JavaScript versions 1.2 and later, also known as JScript, ECMAScript and ECMA-262. Supported by MSIE 5 and later, and many other recent web browsers.
- C, implementing the open source PCRE library. Only match actions are supported.
- Perl, versions 5 and later
- PHP, using either the ereg or preg functions. The ereg functions are always available, but do not support most of the advanced regex syntax. The preg functions are based on PCRE, and require PHP to be compiled with support for this library. The preg functions support the entire regex syntax that RegexBuddy supports.
- Python, using Python's built-in "re" module.
- Ruby, using Ruby's built-in regex operators and classes.
- VB.NET, using the System.Text.RegularExpressions library of the .NET framework.
- Visual Basic 6, using the VBScript RegExp object.
- VBScript, using its built-in RegExp object.

Available Functions

To make it easy for you to apply the same regex techniques in a variety of programming languages, RegexBuddy uses the same function descriptions for all languages it supports. Some functions are not available for all languages, depending on the functionality provided by the language or its regular expression library.

Import regex library: If the regular expression support is not a core part of the language, you need to reference the regex library in your code, before you can use any of the other code snippets.

Test if the regex matches (part of) a string: Tests if the regular expression can be matched anywhere in a subject string, and stores the result in a boolean variable.

Test if the regex matches a string entirely: Tests if the regular expression matches the subject string entirely, and stores the result in a boolean variable. This function is appropriate when validating user input or external data, since you typically want to validate everything you received. If the language does not have a separate function for testing if a string can be matched entirely, RegexBuddy will place anchors around your regular expression to obtain the same effect.

If/else branch whether the regex matches (part of) a string: Tests if the regular expression can be matched anywhere in a subject string. Add code to the *if* section to take action if the match is successful. Add code to the *else* section to take action if the match fails.

If/else branch whether the regex matches a string entirely: Tests if the regex matches a subject string in its entirety, adding anchors to the regex if necessary.

Create an object with details about how the regex matches (part of) a string: Applies the regular expression to a subject string, and returns an object with detailed information about the part of the subject string that could be matched. The details usually include the text that was matched, the offset position in the string where the match started, and the number of characters in the match. If the regex contains capturing groups, the details will also contain the same information for each of the capturing groups. The groups are numbered starting at one, with group number zero being the overall regex match. The Find First button on the Test tab performs the same task as this function.

Note that for regular expressions that consist only of anchors or lookaround, or where everything is optional, it is possible for a successful match to have a length of zero. Such regexes can also match at the point right after the end of the string, in which case the offset points to a non-existent character.

Get the part of a string matched by the regex: Applies the regular expression to a subject string, and returns the part of the subject string that could be matched. This function is appropriate if you are only interested in the first (or only) possible match in the subject string.

Get the part of a string matched by a capturing group: Applies the regular expression to a subject string, and returns the part of the subject string that was matched by a capturing group in the regex. This function is appropriate if you are only interested in a particular part of the first (or only) possible match in the subject string. One of the parameters for this function is the number of the reference to the group. You can easily obtain this number from the regex tree on the Create tab.

Replace all matches in a string: Executes a replace action. All regex matches in the subject string will be substituted with the replacement text. References to capturing groups in the replacement text are also expanded. RegexBuddy automatically converts backslash-style references or dollar-style references if the programming language you selected only supports one style. Unless otherwise indicated in the function's name, the “replace all” function does not modify the original subject string. It returns a new string with the replacements applied.

Split a string: Executes a split action. Returns an array or list of strings from a given subject string. The first string in the array is the part of the subject before the first regex match. The second string is the part between the first and second matches, the third string the part between the second and third matches, etc. The final string is the remainder of the subject after the last regex match. The text actually matched by the regex is not added to the array.

You can specify a limit for the split action. The limit is the maximum number of strings the resulting string or array will contain. In other words, the string is split at most limit-1 times. The last string in the result is the

unsplit remainder of the subject. If the regex can be matched fewer than limit-1 times, the resulting array will have fewer items than your limit.

Create an object to use the same regex for many operations: Before a regular expression can be applied to a subject string, the regex engine needs to convert the textual regex that you (or the user of your software) entered into a binary format that can be processed by the engine's pattern matcher. If you use any of the functions listed above, the regex is (re-)compiled each time you call one of the functions. That is inefficient if you use the same regex over and over. Therefore, you should create an object that stores the regular expression and its associated internal data for the regex engine. Your source code becomes easier to read and maintain if you create regex objects with descriptive names.

Create an object to apply a regex repeatedly to a given string: Some regex libraries, such as Java's java.util.regex package, use a separate pattern matcher object to apply a regex to a particular subject string, instead of using the regex object created by the function above. Explicitly creating and using this object, instead of using one of the convenience functions that create and discard it behind the scenes, makes repeatedly applying the same regex to the same string more efficient. This way you can find all regex matches in the string, rather than just the first one.

Apply the same regex to more than one string: Illustrates how to use the regex object to apply a given regular expression to more than one subject string.

Use regex object to ...: This series of functions creates and uses a regex object to perform its task. The results of these functions are identical to the results of the similarly named functions already described above. The advantage of these functions over the ones above, is that you can reuse the same regex object to use a given regular expression more than once. For the second and following invocations, you obviously only copy and paste the part of the code snippet that uses the regex object into your source code.

Iterate over all matches in a string: Applies and re-applies a regular expression to a subject string, iterating over all matches in the subject, until no further matches can be found. Insert your own code inside the loop to process each match. Match details such as offset and length of the match, the matched text, the results from capturing groups, etc. are available inside the loop.

Iterate over all matches and backreferences in a string: Iterates over all matches in the string like the previous function, and also iterates over all capturing groups or backreferences for each match. Note that the number of available backreferences may be different for each match. If certain capturing groups did not participate at all in the overall match, no details will be available for those groups. In practice, you will usually use the previous function, and only work with the capturing groups that you are specifically interested in. This function illustrates how you can access each group.

Comment with RegexBuddy's regex tree: A series of comment lines with the regular expression “as is” (i.e. not converted to a string or operator) and the regex tree from the Create tab in RegexBuddy.

10. Copying and Pasting Regular Expressions

Copy and pasteis the simple and easy way to transfera regular expression between RegexBuddy and your searching, editing and coding tools. You can use the regular Ctrl+C and Ctrl+V shortcut keys to copy and paste the selected part of the regex as is. Or, you can use the Copy and Paste buttons on RegexBuddy's toolbar to transfer the regular expression in different formats. If you often use RegexBuddy in conjunction with a particular tool or application, you may want to see if it is possible to integrate RegexBuddy with that tool.

The search boxes of text editors, grep utilities, etc., do not require the regular expression to be formatted in any way, beyond being a valid regex pattern. This is exactly the way you enter the regular expression in RegexBuddy, whether you type it in or insert tokens on the Create tab. So you can simply copy and paste the regex “as is”.

If you want to use the regular expression in the source code of an application or script you are developing, the regex needs to be formatted according to the rules of your programming language. Some languages, such as Perl and JavaScript use a special syntax reserved for regular expressions. Other languages rely on external libraries for regular expression support, requiring you to pass regexes to their function calls as strings. This is where things get a bit messy.

Copying The Regex as a String or Operator

In regular expressions, metacharacters must be escaped with a backslash. In many programming languages, backslashes appearing in strings must be escaped with another backslash. This means that the regex «\\» which matches a single backslash, becomes "\\\\" in Java or C/C++. The regex «"\\"» which matches a single backslash between double quotes, becomes "\"\\\\\"“”. How's that for clarity?

When you generate code snippets on the Use tab, RegexBuddy automatically inserts your regexes in the proper format into the code snippets, adapting them to the whims of each language. To use a regex you prepared in RegexBuddy without creating a code snippet, click the Copy button on the main toolbar, above or to the left of the Match/Replace/Split tabs. You can copy the regular expression to the clipboard in one of several formats:

As is: Copies the regex unchanged. Appropriate for tools and applications, but not for source code.

Basic-style string: The style used by programming languages derived from Basic, including Visual Basic. A double-quoted string. A double quote in the regex becomes two double quotes in the string.

C-style string: The style used by programming languages derived from C, including C++, C#, Java, etc. A double-quoted string. Backslashes and double quotes are escaped with a backslash.

Pascal-style string: The style used by programming languages derived from Pascal, including Delphi. A single-quoted string. A single quote in the regex becomes two single quotes in the string.

Perl-style string: The style used by scripting languages such as Perl and PHP, where a double-quoted string is interpolated, but a single-quoted string is not. Quotes used to delimit the string, and backslashes, are escaped with a backslash.

Perl operator: A Perl m/ / operator for match and split actions, and an s/ / / operator for replace actions.

JavaScript operator: A Perl-style / / operator for use with JavaScript. JavaScript uses mode modifiers that differ from Perl's.

Ruby operator: A Perl-style / / operator for use with Ruby. Ruby uses mode modifiers that differ from Perl's.

PHP '/ /' preg string: A Perl-style / / operator in a string for use with PHP's preg functions.

Python string: Unless the regex contains both single and double quote characters, the regex is copied as a Python “raw string”. Raw strings to not require backslashes to be escaped, making regular expressions easier to read. If the regex contains both single and double quotes, the regex is copied as a regular Python string, with quotes and backslashes escaped.

Pasting a String or Operator as The Regular Expression

RegexBuddy can do the opposite conversion when you want to edit a regular expression that is already part of an application's source code. In your source code editor or IDE, select the entire string or operator that holds the regular expression. Make sure quotes and other delimiters are included. Copy it to the clipboard.

Then switch to RegexBuddy, and click the Paste button in the main toolbar. The Paste button's menu offers the same options as the Copy menu, except that they work the other way around. If the clipboard holds the Java string `"\"\\\\\“”`, select “paste as C-style string”, and RegexBuddy will properly interpret the string as the regular expression «"\\"», which matches a single backslash between a pair of double quote characters. When you're done editing the regex, select “copy as C-style string” and you can paste the updated regex as a Java string into your source code.

11. Storing Regular Expressions in Libraries

When you have gone through the effort of creating and testing a regular expression, it would be a waste to use it just once. It makes a lot of sense to store the regexes you create for later reuse, by yourself or by friends and colleagues. RegexBuddy makes storing and sharing regex actions easy.

To store a regular expression, switch to the Library tab in RegexBuddy. If you want to store it into an existing library, click the Open button to select the RegexBuddy Library file. Or click the small downward pointing arrow on the Open button to reopen a library that you recently worked with.

Any changes you make to a RegexBuddy Library are saved automatically. RegexBuddy makes sure that you never lose any regexes you worked so hard on. If you open the same library in more than one instance of RegexBuddy, RegexBuddy makes sure all the instances are kept in sync, by automatically saving and reloading the library.

In other words: you don't have to worry at all about making sure your work is saved. As a precaution against inadvertent modifications, all libraries are opened as "read only" by default. You can toggle the read only state with the check box just below the New and Open buttons on the Library tab's toolbar.

One exception is the sample RegexBuddy library that is included with RegexBuddy. When you open it, RegexBuddy will turn on the “read only” state permanently. The reason is that when you upgrade RegexBuddy, the sample library will be upgraded too. If you want to edit the sample library, click the Save As button to make a copy.

Since RegexBuddy automatically and regularly saves libraries, there is no button or command for you to do so manually. If you want to edit a library, and preserve the original for safekeeping, click the Save As button before turning off the read only option. Save As tells RegexBuddy to create a copy of the library under a new name. Any subsequent changes are automatically saved into the new file. The original file stays the way it is.

Adding a Regex Action to The Library

First, you need to define the regex action in the topmost part of RegexBuddy's window. Then, click on the Library tab's Add button. If the button is disabled, you forgot to turn off “read only”, or you did not type in a regular expression yet.

The entire regex action then appears in the lower right part of RegexBuddy's window. You can further edit it there, and parameterize it when appropriate.

Don't forget to type a clear description in the edit box just below the Library tab's toolbar. You can enter as much text as you want. A good description will greatly help to look up the regex later.

The list at the left shows all regexes in the library, alphabetically sorted by their descriptions. If a regex does not have a description, the regex itself is used instead.

Copying and Pasting Regexes Between Libraries

Each instance of RegexBuddy can open only a single library. If you want to move or duplicate regex actions between two libraries, start a second instance of RegexBuddy. You can easily do that by clicking the New button in the top left corner. Then use the Cut, Copy and Paste buttons to move or copy a regex action from one instance of RegexBuddy to the other.

If you open the same library in more than once RegexBuddy instance, both instances are automatically kept in sync. If you make changes in one instance, they automatically appear in the other instance when you switch to it.

12. Parameterizing a Regular Expression Stored in a Library

When storing a regular expression in a RegexBuddy Library for later reuse, it is sometimes obvious that many variants of the regex should also be in the library. E.g. the regex «`^.{7}(.{3})`» captures the 8th through 10th characters of a line into backreference 1. Though you may want to store this regex for later reuse, it is quite likely that in the future, you will use the same technique to capture a different range of characters.

Rather than storing a large number of similar regexes into a library, you can parameterize a regular expression. After adding the regex to a library, edit it on the Library tab. Replace all parts of the regex that should be variable, with a parameter. A parameters is a sequence of one or more letters A..Z and digits 0..9, delimited by a pair of percentage signs. E.g. turn «`^.{7}(.{3})`» into `^.{%SKIPAMOUNT%}(.{%CAPTUREAMOUNT%})`.

Then click on the Parameters tab. You will see a grid with three columns, and one row for each parameter you inserted into the regex. In the second column, briefly describe the parameter. For lengthy descriptions, use the edit box just below the Library tab's toolbar instead. There, you can enter as much text as you want.

Provide a default value for each parameter in the third column. Substituting each parameter with its default value should result in a valid regular expression. That makes it easier to reuse the regex later.

When substituting a regular expression, RegexBuddy does not interpret a parameter's value in any way. The parameter's tag is simply replaced with the value, whatever it is. Metacharacters in the value are not automatically escaped. Though this means you have to be a bit careful when entering a value for a parameter, it also enables you to substitute a parameter with a complete regular expression. The above example could be generalized into `^.{%SKIPAMOUNT%}(%REGEX%)`. Rather than capturing a fixed number of characters, this regex template will match whichever regular expression you want, starting at a specific column on a line.

When adding a replace action to a library, you can use parameters in the replacement text just like you can use them in the regular expression. If you specify the same parameter more than once, in the regex and/or the replacement text, all occurrences of that parameter will be replaced with the same value.

13. Using a Regular Expression from a Library

RegexBuddy makes it very easy to create regular expressions, which is nice. But it is even nicer to use ready-made regular expressions provided by others, or reuse regexes that you saved yourself in the past.

On the Library tab, click the the Open button to open the RegexBuddy Library from which you want to reuse a regular expression. Or click the small downward pointing arrow on the Open button to reopen a library that you recently worked with. If this is the first time you are using RegexBuddy's Library tab, the list of libraries that you recently worked with will list the standard RegexBuddy library that is included with RegexBuddy itself. This library contains a wide range of sample regex actions that you can use for various purposes.

To use a regex action from the library, select it from the list at the left, and click the Use button. This will copy the regex action to the Match/Replace/Split tabs at the top, where you define regex actions in RegexBuddy. Then you can analyze, test and implement the regular expression as usual.

Substituting Parameters

When storing a regular expression in a RegexBuddy Library, it can be parameterized. When you select a parameterized regular expression and click the Use button, RegexBuddy will ask you what you want to substitute the parameters with. Each parameter has a description that explains what you should enter as the value. A default value is also provided. If you can't read the complete description because the column is too narrow, make the parameter window wider. The description column stretches along with the window.

RegexBuddy does not interpret a parameter's value in any way. The parameter's tag is simply replaced with the value, whatever it is. Metacharacters in the value are not automatically escaped. Though this means you have to be a bit careful when entering a value for a parameter, it also enables you to substitute a parameter with a complete regular expression. You can see a preview of the final regular expression, and the replacement text in case of a replace action, as you type in values for the parameters.

14. GREP: Search and Replace through Files and Folders

If you've defined a match action, the GREP tab enables you to search through any number of files and folders using the regular expression you created. If you defined a replace action, you can use it to search-and-replace through files and folders. This way you can test your regular expression on a larger number of files (rather than just one file on the Test tab), and even perform actual search or file maintenance tasks.

Type the folder you want to search through in the Folders field, or click the ellipsis button next to it to select the folder from a folder tree. If you want to search through multiple folders, simply enter them all delimited by semicolons. Turn on “recurse subfolders” to make RegexBuddy search through all subfolders of the folder(s) you specified.

If you don't want to search through all files in the folder, you can restrict the search using file masks. In a file mask, the asterisk (*) represents any number (including none) of any character, similar to «`.*`» in a regular expression. The question mark (?) represents one single character, similar «`.`» in a regular expression. E.g. the file mask `*.txt` tells RegexBuddy to search through any file with a .txt extension. You can delimit multiple masks with semicolons. To search through all C source and header files, use `*.c;*.h`.

File masks also support a simple character class notation, which matches one character from a list or a range of characters. E.g. to search through all web logs from September 2003, use a file mask such as `www.200309[0123][0-9].log` or `www.200309??.log`.

Sometimes it is easier to specify what you do not want to search through. To do so, mark the "invert mask" option. With this option, a file mask of `*.c;*.h` tells RegexBuddy to search through all files except C source and header files.

By default, RegexBuddy searches through a whole file at once, and allows regex matches to span multiple lines. Traditional grep tools, however, search through files line by line. You can make RegexBuddy do the same by turning on the "line-based" option. When grepping line by line, you can turn on "invert results" to make RegexBuddy list all lines in each file which the regular expression does *not* match.

Target and Backup Files

When searching, RegexBuddy does not modify any files by default. The search matches are simply displayed on screen. To save all the search matches into a single file, select the "save results into a single file" option. Click the ellipsis button to the far right of the target setting to select the file you want to save the results into. To create one file for each file searched through, select "save one file for each searched file". Click the ellipsis button to select the folder to save the files into. Each saved file will have the same name as the file searched through.

When executing a search-and-replace, RegexBuddy can either modify the files it searched through, or create copies of those files and modify the copies. If you select to "copy only modified files“ or ”copy all searched files", click the ellipsis button to select the folder to save the target files into. Again, each target file will have the same name as the original.

RegexBuddy will not prompt you when the GREP action overwrites one or more files. Therefore, you should make sure the backup option is set to anything except "no backups". The "single .bak“ and ”single .~??" options create a backup file for each target file that is overwritten. If the backup file already existed, the

previous backup is lost. The "multi .bak“ and ”multi backup N of" options create numbered backup files without overwriting existing backup files. The "same file name" option keeps one backup for each target file, and gives the backup copy the same name.

The “same file name” backup option requires you to specify a folder into which backup files should be stored. The other backup options allow you to choose. If you specify a backup folder, all backups will be stored in that folder. If not, backups are saved in the same folders as the target files that were overwritten. Keeping backups in a separate folder makes it easy to get rid of the backups later, but makes it harder to find a backup file if you want to restore a file manually.

Preview, Execute and Quick Execute

RegexBuddy can GREP in three modes: preview, execute and quick execute. You can access these modes through the GREP button. A preview displays detailed results in RegexBuddy, without modifying any files. Execute an action to create or modify target files as you specified, and to see detailed results afterwards to verify the results.

Use “quick execute” when you know the GREP action will do what you want, to quickly update files without displaying detailed results in RegexBuddy. The “quick execute” option can be much faster than “execute” when performing a search-and-replace through large numbers of files, since it doesn't require RegexBuddy to keep track of each individual search match. On a typical computer, “preview” and “execute” can handle about 100,000 search matches, while “quick execute” has no practical limits.

Backup Files and Undo

If you followed my advice to have RegexBuddy create backup files, you can instantly undo the effects of a GREP action. Click on the GREP button, and then select Undo from the menu. This will replace all files that were overwritten with their backup copies. The backups are removed in the process. Only the very last action can be undone by RegexBuddy, and only if you do not close RegexBuddy. If you have multiple instances of RegexBuddy open, each instance can undo its own very last GREP action.

If you've verified the results, and all target files are in order, click the GREP button and select Delete Backup Files to delete the backup files created by the very last GREP action.

Opening and Saving Actions and Results

The Clear button clears the GREP action and results. It is not necessary to click the Clear button before starting a new action, but it may make things easier for you by reducing clutter.

Click the Open button to open a GREP action previously saved with the Save button. Note that only the action (regular expression with folder, mask, target and backup settings) is saved into rbg files.

To save the results of an action, click the Export button. When prompted, enter a file name with a .txt or .html extension. If you enter a .html extension, the results are saved into an HTML file. When you view the HTML file in a web browser, it will look just like the results are displayed in RegexBuddy. If you enter any other extension, the results are saved as plain text. If you want to save search results without additional

information such as the number of matches, set the Target option to save the results to file *before* executing the action. Target files only contain search matches, one on each line.

PowerGREP: The Ultimate GREP for Windows

Since RegexBuddy's focus is on creating and testing regular expressions, it only offers basic GREP functionality. While certainly useful, you may want to invest in a powerful GREP tool such as PowerGREP. PowerGREP can search using any number of regexes at once, post-process replacement texts during search-and-replace, arbitrarily section files using an extra set of regular expressions, search through proprietary file formats such as PDF, MS Word and Excel, etc. PowerGREP also keeps a permanent undo history, a feature that can save your day all by itself.

15. Adjust RegexBuddy to Your Preferences

Click the Preferences button to adjust RegexBuddy to your tastes and habits.

Appearance

Large toolbar icons: If you have a high resolution monitor, or simply feel that the icons on RegexBuddy's toolbar are a bit small, turn on “large toolbar icons”.

Place toolbars at the left side instead of at the top: Turn this option on to place toolbars at the left edge of RegexBuddy's window, with the buttons stacked vertically, rather than below the two rows of tabs, with the buttons placed horizontally. Moving the toolbars to the left side conserves quite a bit of screen real estate. Turn on this option if you want to keep RegexBuddy's window small, and still have plenty of room to work. When placed at the left, toolbars cannot show text labels.

Show text labels: Text labels on the toolbar icons make it easier to get started with RegexBuddy. Turn them off if you want to keep RegexBuddy's window small, and some of the buttons become invisible because the labels take up too much space.

Show statusbar: The statusbar displays helpful hints as you move the mouse over different parts of RegexBuddy's window, most importantly the toolbar buttons. Leave this option on as you familiarize yourself with RegexBuddy. Later, you can turn it off to save space on the screen.

Preserve the regular expression and replacement text: When this option is on, you can continue working on the same regular expression when you close and restart RegexBuddy. When off, RegexBuddy will start with a blank regular expression.

Editors

Editors for regular expressions: Configures the match, replace and split edit boxes for defining regular expressions. You can choose whether spaces and tabs should be indicated by small dots and >> signs, and whether line breaks should be indicated by paragraph symbols. Both options are on by default. Click the Choose Font button to select which font you want to use to edit regular expressions. You can choose any font available on your computer.

“Test” and “use” editors: Configures the edit boxes on the Test tab and Use tab. You can choose whether spaces and tabs should be indicated by small dots and >> signs, and whether line breaks should be indicated by paragraph symbols. Both options are off by default. Click the Choose Font button to select which font you want to use to edit regular expressions. You can choose any font available on your computer, though you will probably want to select a monospaced font such as Courier, to make columns line up properly.

Standard or custom cursor: On the Windows platform, you can choose whether you want to use the regular Windows text cursor, or a custom, highly visible text cursor. If you use screen reader software or other accessibility tools that rely on the Windows text cursor to work, choose to use the standard Windows text cursor. Otherwise, using the custom cursor is preferable. The custom cursor is fully adjustable to your taste and eyesight level. On the Linux platform, only the custom cursor is available.

Text cursor blinking style: Choose whether the cursor should always be visible in the “regular” color, whether it should blink on and off using the “regular color”, or whether it should alternate between the “regular” and “alternate” colors. The last option results in the most visible text cursor. Blinking on and off is what the standard windows text cursor does.

Text cursor colors: Configure the colors to go with the cursor's blinking style. Click on the button, and then click on a color in the color palette. You can also navigate the color palette with the arrow keys on the keyboard, or enter custom RGB values below the color palette.

Text cursor shape: “Vertical bar in front of the character” is the way cursors are usually displayed on Windows. “Rectangle covering the character” is the way cursors used to be displayed on text-based interfaces. “Bar or rectangle depending on insert or overwrite mode” is the most useful option. The text cursor then indicates insert or overwrite mode by its shape. You can toggle modes by pressing the Insert key on the keyboard.

Drag cursor color: When you drag-and-drop a piece of text with the mouse, a second cursor will indicate the insertion spot. This cursor is always a static vertical bar. Only its color can be configured.

Regex Colors

Configure the colors used to highlight different parts of regular expressions. You can select one of the preset configurations at the top of the screen. To customize the colors, click on an item in the list of individual colors. Then click the Background Color and Text Color buttons to change the item's colors. You can also change the font style with the bold, italic and underline checkboxes. At the bottom, a sample edit box will show what the color configuration looks like. You can edit the text in the example box to further test the colors.

Highlight Colors

Configure the colors used by the Highlight button on the Test tab. You can select one of the preset configurations at the top of the screen. To customize the colors, click on an item in the list of individual colors. Then click the Background Color and Text Color buttons to change the item's colors. You can also change the font style with the bold, italic and underline checkboxes. At the bottom, a sample edit box will show what the color configuration looks like. You can edit the text in the example box to further test the colors. The regex «`match(.*?(backreference))?`» is used to highlight the sample text.

Programming Colors

Configure the colors used by syntax coloring of the code snippets on the Use tab. You can select one of the preset configurations at the top of the screen. To customize the colors, click on an item in the list of individual colors. Then click the Background Color and Text Color buttons to change the item's colors. You can also change the font style with the bold, italic and underline checkboxes. At the bottom, a sample edit box will show what the color configuration looks like. You can see the effect for several different programming languages. You can edit the text in the example box to further test the colors.

16. Integrate RegexBuddy with Searching, Editing and Coding Tools

RegexBuddy is designed to be used as a companion to whatever software you use regular expressions with. Such applications include search tools, text editing and processing tools, programming and development tools, etc. Whenever you need to write or edit a regular expression while working with those applications, RegexBuddy pops up to provide assistance, and disappears when you finish editing the regex.

Without integrating RegexBuddy with your tools, this workflow takes many steps:

1. In your tool, copy the regular expression to the clipboard.
2. Start RegexBuddy via a desktop or start menu shortcut.
3. Paste the regular expression into RegexBuddy, manually selecting the correct format.
4. Edit the regular expression.
5. Copy the regular expression from RegexBuddy to the clipboard, manually selecting the correct format.
6. Close RegexBuddy.
7. Paste the regular expression into your tool.

Basic Integration

Integrating RegexBuddy with your favorite tools can significantly speed up this workflow by automating most of the steps. The simplest form of integration is by starting RegexBuddy with certain command line parameters. This method is appropriate when you want to integrate RegexBuddy with a 3rd party tool. If the tool has the ability to run external applications, you can easily launch RegexBuddy from within the tool.

E.g. in Microsoft Visual Studio 2003, you could select Tools|External Tools from the menu, and specify the following arguments: `-getfromclipboard -putonclipboard -basic -appname "Visual Studio"`. Use `-c` instead of `-basic` if you develop in C++, C# or J# rather than Visual Basic.

When adding RegexBuddy to Borland Delphi's Tools menu, you can specify `-getfromclipboard -putonclipboard -delphi -appname Delphi` for the parameters.

This style of integration, which only takes a few minutes to implement, reduces the workflow to:

1. In your tool, copy the regular expression to the clipboard.
2. Start RegexBuddy from a menu item, button or keyboard shortcut in your tool.
3. Edit the regular expression, which RegexBuddy automatically grabs from the clipboard, in the right format.
4. Click the Send To button in RegexBuddy, which closes RegexBuddy and automatically puts the modified regex on the clipboard, in the right format.
5. Paste the regular expression into your tool.

Though you still need to copy and paste the regular expression from and into your tool, the reduced workflow is a lot more convenient. You can launch RegexBuddy form within your tool, and you no longer need to copy and paste the regex in RegexBuddy.

Advanced Integration

Tighter integration is possible with tools that have a programming interface (API), and with software that you develop yourself. On the Microsoft Windows platform, RegexBuddy provides a COM Automation interface, which you can easily import and call from any development tool or language that supports COM. The interface enables you to launch RegexBuddy, and send and receive regex actions in response to the user's actions.

On the Linux platform, you can launch RegexBuddy through fork() and exec(). You can then send and receive regex actions via standard input and standard output pipes.

Though the implementation is quite different because of platform restrictions and standards, the end result is the same. Except for actually editing the regular expression, the workflow is automated entirely:

1. Start RegexBuddy from a menu item, button or keyboard shortcut in your tool. This automatically sends the regex to RegexBuddy.
2. Edit the regular expression.
3. Click the Send To button in RegexBuddy, which automatically updates the regex in your tool, and closes RegexBuddy.

If you are a software developer, and some of your products support regular expressions, make your customers happy and provide tight integration with RegexBuddy. Take a look at PowerGREP and EditPad Pro to see how convenient they make it to edit a regular expression with RegexBuddy. If you want to promote or sell RegexBuddy to your customers, please drop us an email. We're happy to discuss a mutually beneficial partnership with you or your company.

17. Basic Integration with RegexBuddy

Integrating RegexBuddy with your favorite tools can significantly speed up this workflow by automating most of the steps. The simplest form of integration is by starting RegexBuddy with certain command line parameters. This method is appropriate when you want to integrate RegexBuddy with a 3rd party tool. If the tool has the ability to run external applications, you can easily launch RegexBuddy from within the tool.

RegexBuddy's command line parameters are case insensitive. The order in which they appear is irrelevant, except for parameters that must be followed by a second “data” parameter. The second parameter must immediately follow the first parameter, separated with a space. Values with spaces in them must be delimited by a pair of double quotes. On Linux, parameters start with a single hyphen. On Windows, parameters can start with a hyphen or a forward slash.

`-getfromclipboard`

> RegexBuddy will use the text on the clipboard as the regular expression. The text is used “as is”, unless you specify one of the format options.

`-putonclipboard`

> RegexBuddy will show the Send To button. When clicked, the regex is copied to the clipboard. The regex is copied “as is”, unless you specify one of the format options.

`-c`

> Use the “C-style string” format for both `-getfromclipboard` and `-putonclipboard`.

`-basic`

> Use the “Basic-style string” format for both `-getfromclipboard` and `-putonclipboard`.

`-pascal`

> Use the “Pascal-style string” format for both `-getfromclipboard` and `-putonclipboard`.

`-perl`

> Use the “Perl-style string” format for both `-getfromclipboard` and `-putonclipboard`.

`-perlop`

> Use the "Perl m/ / operator" format for both `-getfromclipboard` and `-putonclipboard`.

`-javascript`

> Use the "Perl-style / / operator" format for both `-getfromclipboard` and `-putonclipboard`.

`-preg`

> Use the "PHP '/ /' preg string" format for both `-getfromclipboard` and `-putonclipboard`.

`-appname “Application Name”`

The `-appname` parameter must be immediately followed with a parameter that indicates the name of the application that is invoking RegexBuddy. RegexBuddy will show this name in its caption bar, to make it clear which application the Regex will be sent to. You should always specify `-appname` when using `-putonclipboard`, or when using standard input and standard output pipes.

`-library filename.rbl`

Opens the specified library on the Library tab. Use this parameter if you want to associate RegexBuddy with rbl files.

`-grep filename.rbg`

Opens the specified GREP action on the GREP tab. Use this parameter if you want to associate RegexBuddy with rbg files.

`-activate 1234`

This parameter is only available on the Windows platform. Use it in combination with `-putonclipboard`. `-putonclipboard` must be followed by a decimal unsigned integer that indicates the window handle of the active top-level form in your application. When the Send To button is clicked, RegexBuddy will use the SetForeground() Win32 API call to bring your application to the front. That way, the Sent To button enables you to continue working with that application right away.

`-stdio`

This parameter is only available on the Linux platform. It enables communication between your application and RegexBuddy using standard input and standard output pipes.

Important: Do not specify the `-stdio` parameter unless you will actually connect a pipe to RegexBuddy's standard input, and close that pipe after sending data. Otherwise, RegexBuddy will wait forever for the pipe to be closed.

18. Integrating RegexBuddy using Standard Input and Output

If you are a software developer, and some of your products support regular expressions, make your customers happy and provide tight integration with RegexBuddy. That way, they can edit the regexes they use in your software with RegexBuddy at the click of a button, without having to manually copy and paste regexes between your software and RegexBuddy.

On the Linux platform, you can launch RegexBuddy through fork() and exec(). You can then send and receive regex actions via standard input and standard output pipes.

Take a look at EditPad Pro to see how convenient they make it to edit a regular expression with RegexBuddy. If you want to promote or sell RegexBuddy to your customers, please drop us an email. We're happy to discuss a mutually beneficial partnership with you or your company.

How RegexBuddy Communicates via Standard Input and Output

When you launch RegexBuddy with the `-stdio` command line parameter, RegexBuddy will read the regular expression or regex action from standard input before showing itself. If you fail to send a proper regular expression or regex action structure (see below) to standard input, RegexBuddy will wait until the pipe is closed. So make sure that your application closes its end the pipe that it connected to RegexBuddy's standard input, before your application starts waiting for any output from RegexBuddy. Otherwise, deadlock will occur.

After receiving the regex action from your application, RegexBuddy will appear, with your application's regex action ready for editing. The Send To button will appear in the toolbar. When clicked, the Send To button will send the modified regex action to your application via standard output. RegexBuddy will use the same format as you used to send the original action via standard input. RegexBuddy will then close its end of the pipe, and itself.

If the user closes RegexBuddy without clicking the Send To button, RegexBuddy will close its end of the standard output pipe without sending any data to it. When that happens, your application should continue using the original regex action.

Sending and Receiving The Regular Expression

You can send and receive the regular expression in two ways. The simple way is to send the regular expression only, as plain 8-bit text. The other way is to send a complete regex action structure.

When sending the regex as text, simply send it as 8-bit characters (i.e. no Unicode or UTF-8). Do *not* send a terminating NULL or EOF character. Just close your application's end of the pipe after writing the regex to it.

By default, RegexBuddy will use the text you send “as is”. If you want RegexBuddy to interpret the text you send in a particular way, use one of the formatting command line parameters. E.g. specify `-c` if you will send

a double-quoted C-style string. RegexBuddy will unquote and unescape the string, since the quotes are not part of the regular expression.

RegexBuddy will send back the modified regex in the same way. It will send the regex back as plain 8-bit text without any terminating character, closing its end of the pipe when all characters have been send. If you specified a formatting command line parameter, RegexBuddy will send back the regex using that formatting.

RegexBuddy Action Binary Structure

If you want to send a complete RegexBuddy action, that is a regular expression with assorted options, you need to send a RegexBuddy Action binary structure. RegexBuddy uses the first byte of the data you send to determine the format. If it is a byte 0x01, then RegexBuddy assumes you are sending a complete structure. If not, it assumes you are sending the regex only as plain text (and the first byte is the first character in that text).

The structure has the following format. The strings use 8-bit characters, and are NOT null-terminated.

```
1 byte          Version number.  Must be 1 or 2.
1 byte          Kind of action; 0 = match; 1 = replace; 2 = split
2 bytes         Length of the regex (can be zero)
n chars         Regex string
2 bytes         Length of the replacement (can be zero)
n chars         Replacement string
1 byte          Dot matches all (0 = off; 1 = on)
1 byte          Case insensitive (0 = off; 1 = on)
1 byte          ^ and $ match at line breaks (0 = off; 1 = on)
4 bytes         Split limit
2 bytes         Length of the description (can be zero)
n chars         Description string
2 bytes         Number of parameters (can be zero)

For each parameter

        2 bytes         Length of the parameter's ID
        n chars         Parameter ID string (including % signs)
        2 bytes         Length of the parameter's description
        n chars         Description string
        2 bytes         Length of the parameter's default value
        n chars         Default value string
```

In practice, you will not send a description or parameters. You will simply send 0x0000 as the length of the description, and 0x0000 as the number of parameters, and then close your application's end of the pipe.

When reading back the modified regex, it is possible that RegexBuddy will send a description and parameters. That can happen if the user used a regular expression from a library. The description and parameters do not have any meaning outside a library.

The reason why RegexBuddy includes this data, is that the structure used for standard input and output is the same structure as used for copying and pasting regex actions on the Library tab. RegexBuddy uses the MIME type `application/regexbuddy.action` to identify the action on the clipboard. Note that on the clipboard, future versions of RegexBuddy may use a different version of the structure. If the version number is not 1, you cannot assume anything about the data. Tell the user to upgrade his or her copy of your

software. This is not an issue when using standard input and output. RegexBuddy will always send back the regex action using the same version of the structure as your application used when sending it.

Note about Strings and Versions

In RegexBuddy version 1 binary structures, “string” is an ANSI string (single or double byte). The length of a string is expressed in bytes. In RegexBuddy version 2 binary structures, “string” is a UCS-2 Unicode string. The length of a string is expressed in the number of UCS-2 code points.

RegexBuddy version 1 and version 2 binary structures use exactly the same structure, except for the type of string used (ANSI or UCS-2). RegexBuddy 1.x.x reads and writes version 1 binary structures only. RegexBuddy 2.x.x can read both version 1 and 2 binary structures from standard input, and will write the same version back to standard output.

19. Integrating RegexBuddy Using COM Automation

If you are a software developer, and some of your products support regular expressions, make your customers happy and provide tight integration with RegexBuddy. That way, they can edit the regexes they use in your software with RegexBuddy at the click of a button, without having to manually copy and paste regexes between your software and RegexBuddy.

On the Microsoft Windows platform, RegexBuddy provides a COM Automation interface, which you can easily import and call from any development tool or language that supports COM. The interface enables you to launch RegexBuddy, and send and receive regex actions in response to the user's actions. It is a single instance interface, which means that each application has its own private instance of RegexBuddy.

Take a look at PowerGREP and EditPad Pro to see how convenient they make it to edit a regular expression with RegexBuddy. If you want to promote or sell RegexBuddy to your customers, please drop us an email. We're happy to discuss a mutually beneficial partnership with you or your company.

Demo Applications Implementing RegexBuddy's COM Automation

In the Clients subfolder of the folder where you installed RegexBuddy (C:\Program Files\JGsoft\RegexBuddy by default), you will find four sample applications that communicate with RegexBuddy through its COM Automation interface. Two are written in Borland Delphi (Win32), and the other two are written in C# (.NET). The C# samples include a .bdsproj file for Borland C#Builder, and a .csproj file for Visual Studio.

Importing RegexBuddy's Type Library

RegexBuddy automatically registers its automation interface with Windows. So if RegexBuddy has been run at least once on a computer, the automation interface is available. To automate RegexBuddy via COM, you need to import its type library. It is stored in RegexBuddy.exe, which is installed under C:\Program Files\JGsoft\RegexBuddy by default.

In Borland Delphi (Win32), select Project|Import Type Library from the menu. Import “RegexBuddy API”, and install the unit into a package. A new component called TRegexBuddyIntf appears on the component palette. This component implements the methods and events you can use to communicate with RegexBuddy. Drop this component on a form or data module. Set ConnectKind to ckNewInstance, and make sure AutoConnect is False. This will ensure your application has its own instance of RegexBuddy, and that RegexBuddy only appears when the user actually wants to edit a regular expression. Call the component's Connect() method the first time the user wants to edit a regular expression with RegexBuddy. For efficiency, do not call the Disconnect() method until your application terminates (at which point it will be called automatically). Assign an event handler either to OnFinishRegex or OnFinishAction, depending on which way of communication you prefer (see below). Your application should only use once instance of the TRegexBuddyIntf component. Otherwise, it will launch multiple instances of RegexBuddy. Doing so would not cause any errors, but does waste resources.

In Visual Studio.NET, right-click on “References” in the Solution Explorer, and pick “Add Reference”. Switch to the COM tab, and choose “RegexBuddy API”. In Borland C#Builder, do the same from the Project Manager. After adding the reference, import the RegexBuddy namespace. Then you can easily access

the RegexBuddyIntfClass class. Create a new object from this class the first time the user wants to edit a regular expression with RegexBuddy. Use the same object for all subsequent times the user wants to edit the same or another regex. Do not delete the object until your application terminates. Each time you create an object of RegexBuddyIntfClass, a new RegexBuddy instance is launched. You also need to create an instance of either IRegexBuddyIntfEvents_FinishRegexEventHandler or IRegexBuddyIntfEvents_FinishActionEventHandler and assign it to the FinishRegex or FinishAction event of your RegexBuddyIntfClass object. Do not assign both. Only assign the one you will actually use.

Note that to successfully communicate with RegexBuddy, two-way communication is required. Your application not only needs to call the methods of the RegexBuddyIntf interface to send the regular expression to RegexBuddy. It also needs to implement an event sink for either the FinishRegex() or FinishAction() method defined in the IRegexBuddyIntfEvents interface. Not all development and scripting tools that can call COM automation objects can also implement event sinks. The tool must support “early binding”.

RegexBuddyIntf Interface

The RegexBuddyIntf COM automation interface provides the following methods:

```
void IndicateApp(BSTR AppName, uint Wnd)
```

Call `IndicateApp` right after connecting to the COM interface. `AppName` will be displayed in RegexBuddy's caption bar, to make it clear that this instance is connected to your application. `Wnd` is the handle of your application's top-level form where the regex is used. RegexBuddy will call `SetForegroundWindow(Wnd)` to activate your application when the user is done with RegexBuddy (i.e. after a call to `FinishAction` or `FinishRegex`). You can call `IndicateApp` as often as you want, as long as you also call it right after connecting to the COM interface.

```
void InitRegex(BSTR Regex, uint StringType)
```

Call `InitRegex` to make RegexBuddy show up with the given regular expression. RegexBuddy will follow up with a call to `FinishRegex` when the user closes RegexBuddy by clicking the Send To button. `Regex` is the regular expression that the user will edit in RegexBuddy. Can be an empty string. `StringType` tells RegexBuddy how to interpret the string you provided:

0: Use the regex as is.
1: The regex is passed as a C-style string. (C, C++, C#, Java, etc.)
2: The regex is passed as a Pascal-style string. (Pascal, Delphi, Kylix, etc.)
3: The regex is passes as a Perl-style string. (Perl, PHP, etc.)
4: The regex is passed as a Perl m/ / or s/ / / operator.
5: The regex is passed as a Basic-style string. (Visual Basic, etc.)
6: The regex is passed as a Perl-style / / operator. (JavaScript, Ruby, etc.)
7: The regex is passed as a PHP '/ /' preg string.

By using a `StringType` other than zero, you can let RegexBuddy take care of adding and removing quote characters, escaping characters, etc. Other values of StringType are reserved for future expansion.

```
void InitAction(VARIANT Action)
```

Call `InitAction` to make RegexBuddy show up with the given RegexBuddy Action. RegexBuddy will follow up with a call to `FinishAction` when the user closes RegexBuddy by clicking the Send To button.

```
BOOL CheckActionVersion(uint Version)
```

Returns TRUE if RegexBuddy supports this version of the RegexBuddy Action COM Variant Structure. Currently only version 1 is supported. This function is made available to allow for future expansion of this structure. If in the future multiple versions of the structure, you will be able to call `CheckActionVersion` to determine if the user's copy of RegexBuddy supports the newer version. This way you can make your application backward compatible with older versions of RegexBuddy, or simply tell the user to upgrade his or her copy of RegexBuddy. Note that `InitAction` and `FinishAction` are forward compatible. If you only support version 1 of the variant structure, you do not need to call `CheckActionVersion`, since version 1 is always supported.

```
uint GetWindowHandle()
```

Returns the window handle of RegexBuddy's main form. Pass this to `SetForegroundWindow()` after calling `InitRegex` or `InitAction`. `SetForegroundWindow()` only works when called by the thread that has input focus, so RegexBuddy cannot bring itself to front. That is your application's duty. (Likewise, RegexBuddy will bring your application to front after calling `FinishAction` or `FinishRegex`, using the window handle you passed on in `IndicateApp()`.)

RegexBuddyIntfEvents Interface

```
FinishRegex(BSTR Regex, uint StringType)
```

You must provide an event handler for this event if you call `InitRegex`. After calling `InitRegex`, RegexBuddy will call `FinishRegex` when the user closes RegexBuddy by clicking the Send To button. RegexBuddy will provide the final regular expression, using the same string type as you used in the last call to `InitRegex`.

```
FinishAction(VARIANT Action)
```

You must provide an event handler for this event if you call `InitAction`. After calling `InitAction`, RegexBuddy will call `FinishAction` when the user closes RegexBuddy by clicking the Send To button.

Which Methods to Call and Events to Handle

In summary, you must call `IndicateApp` immediately after connecting to RegexBuddy. Then, call either `InitRegex` or `InitAction` each time the user wants to edit a regular expression with RegexBuddy. RegexBuddy follows up a call to `InitRegex` or `InitAction` with a corresponding call to `FinishRegex` or `FinishAction`. If you call `InitRegex` or `InitAction` again before receiving a call to `FinishRegex` or `FinishAction`, the effects of the previous call to `InitRegex` or `InitAction` are canceled.

If the user closes RegexBuddy without clicking the Send To button, your application will not receive a call to `FinishRegex` or `FinishAction`.

RegexBuddy Action COM Variant Structure

The `InitAction` method and the `FinishAction` event handler of RegexBuddy's COM interface pass the RegexBuddy Action as a Variant array. The array consists of the following elements:

```
0: int  Version number.  Must be set to 1.
        Future versions of RegexBuddy may change the format of the variant array.
        Specifying the version number makes your application forward compatible.
        RegexBuddy will use the same format when calling FinishAction as you used
        when calling InitAction.
1: int  Kind of action; 0 = match; 1 = replace; 2 = split
2: BSTR The regular expression.
3: BSTR The replacement text.
4: BOOL Dot matches all
5: BOOL Case insensitive
6: BOOL ^ and $ match at line breaks
7: int  Split limit (if >= 2, the split array contains at most  items)
8: BSTR Description of the action.
        Used as the default description if the user adds the action to a library.
```

All elements must be present in the array, even those that are not applicable given the kind of action (i.e. element 3 only applies in a replace action, and 7 only applies in a split action). The values that do not apply are used as the defaults if the user changes the kind of action in RegexBuddy. RegexBuddy may also fill elements that do not apply when calling FinishAction. You can ignore those if you want.

20. Contact RegexBuddy's Developer and Publisher

RegexBuddy is developed and published by JGsoft - Just Great Software.

For the latest information on RegexBuddy, please visit the official web site at http://www.regexbuddy.com/.

If you have purchased RegexBuddy, you are entitled to free technical support via email. The technical support only covers the installation and use of RegexBuddy itself. In particular, technical support does not cover learning and using regular expressions.

Before requesting technical support, please use the Check New Version command in the Help menu to see if you are using the latest version of RegexBuddy. We take pride in quickly fixing bugs and resolving problems in free minor updates. If you encounter a problem with RegexBuddy, it is quite possible that we have already released a new version that no longer has this problem.

To request technical support, please use the Support and Feedback command in the Help menu. This command will show some basic information about your computer and your copy of RegexBuddy. Please copy and paste this information into your email, as it will help us to respond more quickly to your inquiry. If the problem is that you are unable to run RegexBuddy, and thus cannot access the Support and Feedback command, you can email support@regexbuddy.com.

If you have any comments about RegexBuddy, good or bad, suggestions for improvements, please do not hesitate to send them to our technical support department. While we cannot implement each and every user wish, we do take all feedback into account when developing new versions of our software. Customer feedback is an essential part of Just Great Software.

To buy a single user or site license to RegexBuddy, please visit http://www.regexbuddy.com/buynow.html for a complete list of current purchasing options, and up to date pricing information. If you have any questions about buying RegexBuddy not answered on that page, please contact sales@regexbuddy.com.

Part 2

Regular Expression Tutorial

1. Regular Expression Tutorial

In this tutorial, I will teach you all you need to know to be able to craft powerful time-saving regular expressions. I will start with the most basic concepts, so that you can follow this tutorial even if you know nothing at all about regular expressions yet.

But I will not stop there. I will also explain how a regular expression engine works on the inside, and alert you at the consequences. This will help you to understand quickly why a particular regex does not do what you initially expected. It will save you lots of guesswork and head scratching when you need to write more complex regexes.

What Regular Expressions Are Exactly - Terminology

Basically, a regular expression is a pattern describing a certain amount of text. Their name comes from the mathematical theory on which they are based. But we will not dig into that. Since most people including myself are lazy to type, you will usually find the name abbreviated to regex or regexp. I prefer regex, because it is easy to pronounce the plural "regexes". In this book, regular expressions are printed guillemots: «`regex`». They clearly separate the pattern from the surrounding text and punctuation.

This first example is actually a perfectly valid regex. It is the most basic pattern, simply matching the literal text „`regex`”. A "match" is the piece of text, or sequence of bytes or characters that pattern was found to correspond to by the regex processing software. Matches are indicated by double quotation marks, with the left one at the base of the line.

«`\b[A-Z0-9._%-]+@[A-Z0-9-]+\.[A-Z]{2,4}\b`» is a more complex pattern. It describes a series of letters, digits, dots, underscores, percentage signs and hyphens, followed by an at sign, followed by another series of letters, digits and hyphens, finally followed by a single dot and between two and four letters. In other words: this pattern describes an email address.

With the above regular expression pattern, you can search through a text file to find email addresses, or verify if a given string looks like an email address. In this tutorial, I will use the term "string" to indicate the text that I am applying the regular expression to. I will indicate strings using regular double quotes. The term “string” or “character string” is used by programmers to indicate a sequence of characters. In practice, you can use regular expressions with whatever data you can access using the application or programming language you are working with.

Different Regular Expression Engines

A regular expression “engine” is a piece of software that can process regular expressions, trying to match the pattern to the given string. Usually, the engine is part of a larger application and you do not access the engine directly. Rather, the application will invoke it for you when needed, making sure the right regular expression is applied to the right file or data.

As usual in the software world, different regular expression engines are not fully compatible with each other. It is not possible to describe every kind of engine and regular expression syntax (or “flavor”) in this tutorial. I will focus on the regex flavor used by Perl 5, for the simple reason that this regex flavor is the most popular

one, and deservedly so. Many more recent regex engines are very similar, but not identical, to the one of Perl 5. Examples are the open source PCRE engine (used in many tools and languages like PHP), the .NET regular expression library, and the regular expression package included with version 1.4 and later of the Java JDK. I will point out to you whenever differences in regex flavors are important, and which features are specific to the Perl-derivatives mentioned above.

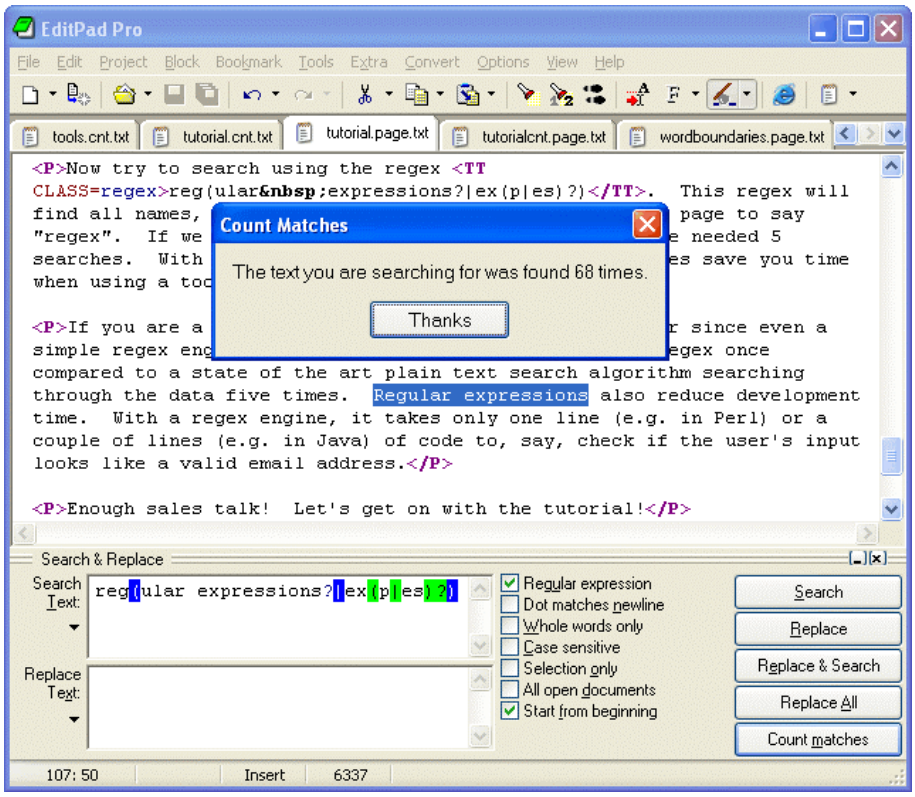
Give Regexes a First Try

You can easily try the following yourself in a text editor that supports regular expressions, such as EditPad Pro. If you do not have such an editor, you can download the free evaluation version of EditPad Pro to try this out. EditPad Pro's regex engine is fully functional in the demo version. As a quick test, copy and paste the text of this page into EditPad Pro. Then select Edit|Search and Replace from the menu. In the search pane that appears near the bottom, type in «`regex`» in the box labeled “Search Text”. Mark the “Regular expression” checkbox, unmark “All open documents” and mark “Start from beginning”. Then click the Search button and see how EditPad Pro's regex engine finds the first match. When “Start from beginning” is checked, EditPad Pro uses the entire file as the string to try to match the regex to.

When the regex has been matched, EditPad Pro will automatically turn off “Start from beginning”. When you click the Search button again, the remainder of the file, after the highlighted match, is used as the string. When the regex can no longer match the remaining text, you will be notified, and “Start from beginning” is automatically turned on again.

Now try to search using the regex «`reg(ular expressions?|ex(p|es)?)`» . This regex will find all names, singular and plural, I have used on this page to say “regex”. If we only had plain text search, we would have needed 5 searches. With regexes, we need just one search. Regexes save you time when using a tool like EditPad Pro.

If you are a programmer, your software will run faster since even a simple regex engine applying the above regex once will outperform a state of the art plain text search algorithm searching through the data five times. Regular expressions also reduce development time. With a regex engine, it takes only one line (e.g. in Perl, PHP, Java or .NET) or a couple of lines (e.g. in C using PCRE) of code to, say, check if the user's input looks like a valid email address.



2. Literal Characters

The most basic regular expression consists of a single literal character, e.g.: «a». It will match the first occurrence of that character in the string. If the string is "Jack is a boy", it will match the „a" after the "J". The fact that this "a" is in the middle of the word does not matter to the regex engine. If it matters to you, you will need to tell that to the regex engine by using word boundaries. We will get to that later.

This regex can match the second „a" too. It will only do so when you tell the regex engine to start searching through the string after the first match. In a text editor, you can do so by using its "Find Next" or "Search Forward" function. In a programming language, there is usually a separate function that you can call to continue searching through the string after the previous match.

Similarly, the regex «cat» will match „cat" in "About cats and dogs". This regular expression consists of a series of three literal characters. This is like saying to the regex engine: find a «c», immediately followed by an «a», immediately followed by a «t».

Note that regex engines are case sensitive by default. «cat» does not match "Cat", unless you tell the regex engine to ignore differences in case.

Special Characters

Because we want to do more than simply search for literal pieces of text, we need to reserve certain characters for special use. In the regex flavors discussed in this tutorial, there are 11 characters with special meanings: the opening square bracket «[», the backslash «\», the caret «^», the dollar sign «$», the period or dot «.», the vertical bar or pipe symbol «|», the question mark «?», the asterisk or star «*», the plus sign «+», the opening round bracket «(» and the closing round bracket «)». These special characters are often called "metacharacters".

If you want to use any of these characters as a literal in a regex, you need to escape them with a backslash. If you want to match „1+1=2", the correct regex is «1\+1=2». Otherwise, the plus sign will have a special meaning.

Note that «1+1=2», with the backslash omitted, is a valid regex. So you will not get an error message. But it will not match "1+1=2". It would match „111=2" in "123+111=234", due to the special meaning of the plus character.

If you forget to escape a special character where its use is not allowed, such as in «+1», then you will get an error message.

All other characters should not be escaped with a backslash. That is because the backslash is also a special character. The backslash in combination with a literal character can create a regex token with a special meaning. E.g. «\d» will match a single digit from 0 to 9.

Special Characters and Programming Languages

If you are a programmer, you may be surprised that characters like the single quote and double quote are not special characters. That is correct. When using a regular expression or grep tool like PowerGREP or the

search function of a text editor like EditPad Pro, you should not escape or repeat the quote characters like you do in a programming language.

In your source code, you have to keep in mind which characters get special treatment inside strings by your programming language. That is because those characters will be processed by the compiler, before the regex library sees the string. So the regex «`1\+1=2`» must be written as `"1\\+1=2"` in C++ code. The C++ compiler will turn the escaped backslash in the source code into a single backslash in the string that is passed on to the regex library. To match „`c:\temp`”, you need to use the regex «`c:\\temp`». As a string in C++ source code, this regex becomes `"c:\\\\temp"`. Four backslashes to match a single one indeed.

See the tools and languages section in this book for more information on how to use regular expressions in various programming languages.

Non-Printable Characters

You can use special character sequences to put non-printable characters in your regular expression. Use «`\t`» to match a tab character (ASCII 0x09), «`\r`» for carriage return (0x0D) and «`\n`» for line feed (0x0A). More exotic non-printables are «`\a`» (bell, 0x07), «`\e`» (escape, 0x1B), «`\f`» (form feed, 0x0C) and «`\v`» (vertical tab, 0x0B). Remember that Windows text files use “`\r\n`” to terminate lines, while UNIX text files use “`\n`”.

You can include any character in your regular expression if you know its hexadecimal ASCII or ANSI code for the character set that you are working with. In the Latin-1 character set, the copyright symbol is character 0xA9. So to search for the copyright symbol, you can use «`\xA9`». Another way to search for a tab is to use «`\x09`». Note that the leading zero is required.

If your regular expression engine supports Unicode, use «`\uFFFF`» rather than «`\xFF`» to insert a Unicode character. The euro currency sign occupies code point 0x20A0. If you cannot type it on your keyboard, you can insert it into a regular expression with «`\u20A0`».

3. First Look at How a Regex Engine Works Internally

Knowing how the regex engineworks will enable you to craft better regexes more easily. It will help you understand quickly why a particular regex does not do what you initially expected. This will save you lots of guesswork and head scratching when you need to write more complex regexes.

There are two kinds of regular expression engines: text-directed engines, and regex-directed engines. Jeffrey Friedl calls them DFA and NFA engines, respectively. All the regex flavors treated in this tutorial are based on regex-directed engines. This is because certain very useful features, such as lazy quantifiers and backreferences, can only be implemented in regex-directed engines. No surprise that this kind of engine is more popular.

Notable tools that use text-directed engines are awk, egrep, flex, lex, MySQL and Procmail. For awk and egrep, there are a few versions of these tools that use a regex-directed engine.

You can easily find out whether the regex flavor you intend to use has a text-directed or regex-directed engine. If backreferences and/or lazy quantifiers are available, you can be certain the engine is regex-directed. You can do the test by applying the regex «`regex|regex not`» to the string "`regex not`". If the resulting match is only „`regex`", the engine is regex-directed. If the result is „`regex not`", then it is text-directed. The reason behind this is that the regex-directed engine is "eager".

In this tutorial, after introducing a new regex token, I will explain step by step how the regex engine actually processes that token. This inside look may seem a bit long-winded at certain times. But understanding how the regex engine works will enable you to use its full power and help you avoid common mistakes.

The Regex-Directed Engine Always Returns the Leftmost Match

This is a very important point to understand: a regex-directed engine will always return the leftmost match, even if a "better" match could be found later. When applying a regex to a string, the engine will start at the first character of the string. It will try all possible permutations of the regular expression at the first character. Only if all possibilities have been tried and found to fail, will the engine continue with the second character in the text. Again, it will try all possible permutations of the regex, in exactly the same order. The result is that the regex-directed engine will return the *leftmost* match.

When applying «`cat`» to "`He captured a catfish for his cat.`", the engine will try to match the first token in the regex «`c`» to the first character in the match "`H`". This fails. There are no other possible permutations of this regex, because it merely consists of a sequence of literal characters. So the regex engine tries to match the «`c`» with the "`e`". This fails too, as does matching the «`c`» with the space. Arriving at the 4th character in the match, «`c`» matches „`c`". The engine will then try to match the second token «`a`» to the 5th character, „`a`". This succeeds too. But then, «`t`» fails to match "`p`". At that point, the engine knows the regex cannot be matched starting at the 4th character in the match. So it will continue with the 5th: "`a`". Again, «`c`» fails to match here and the engine carries on. At the 15th character in the match, «`c`» again matches „`c`". The engine then proceeds to attempt to match the remainder of the regex at character 15 and finds that «`a`» matches „`a`" and «`t`» matches „`t`".

The entire regular expression could be matched starting at character 15. The engine is "eager" to report a match. It will therefore report the first three letters of catfish as a valid match. The engine never proceeds beyond this point to see if there are any "better" matches. The first match is considered good enough.

In this first example of the engine's internals, our regex engine simply appears to work like a regular text search routine. A text-directed engine would have returned the same result too. However, it is important that you can follow the steps the engine takes in your mind. In following examples, the way the engine works will have a profound impact on the matches it will find. Some of the results may be surprising. But they are always logical and predetermined, once you know how the engine works.

4. Character Classes or Character Sets

With a "character class", also called “character set”, you can tell the regex engine to match only one out of several characters. Simply place the characters you want to match between square brackets. If you want to match an a or an e, use «`[ae]`». You could use this in «`gr[ae]y`» to match either „`gray`” or „`grey`”. Very useful if you do not know whether the document you are searching through is written in American or British English.

A character class matches only a single character. «`gr[ae]y`» will not match “`graay`”, “`graey`” or any such thing. The order of the characters inside a character class does not matter. The results are identical.

You can use a hyphen inside a character class to specify a range of characters. «`[0-9]`» matches a *single* digit between 0 and 9. You can use more than one range. «`[0-9a-fA-F]`» matches a single hexadecimal digit, case insensitively. You can combine ranges and single characters. «`[0-9a-fxA-FX]`» matches a hexadecimal digit or the letter X. Again, the order of the characters and the ranges does not matter.

Useful Applications

Find a word, even if it is misspelled, such as «`sep[ae]r[ae]te`» or «`li[cs]en[cs]e`». Find an identifier in a programming language with «`[A-Za-z_][A-Za-z_0-9]*`». Find a C-style hexadecimal number with «`0[xX][A-Fa-f0-9]+`».

Negated Character Classes

Typing a caret after the opening square bracket will negate the character class. The result is that the character class will match any character that is *not* in the character class. Unlike the dot, negated character classes also match (invisible) line break characters.

It is important to remember that a negated character class still must match a character. «`q[^u]`» does *not* mean: “a q not followed by a u”. It means: “a q followed by a character that is not a u”. It will not match the q in the string “`Iraq`”. It will match the q and the space after the q in “`Iraq is a country`”. Indeed: the space will be part of the overall match, because it is the “character that is not a u” that is matched by the negated character class in the above regexp. If you want the regex to match the q, and only the q, in both strings, you need to use negative lookahead: «`q(?!u)`». But we will get to that later.

Metacharacters Inside Character Classes

Note that the only special characters or metacharacters inside a character class are the closing bracket (]), the backslash (\), the caret (^) and the hyphen (-). The usual metacharacters are normal characters inside a character class, and do not need to be escaped by a backslash. To search for a star or plus, use «`[+*]`». Your regex will work fine if you escape the regular metacharacters inside a character class, but doing so significantly reduces readability.

To include a backslash as a character without any special meaning inside a character class, you have to escape it with another backslash. «`[\\x]`» matches a backslash or an x. The closing bracket (]), the caret (^) and the

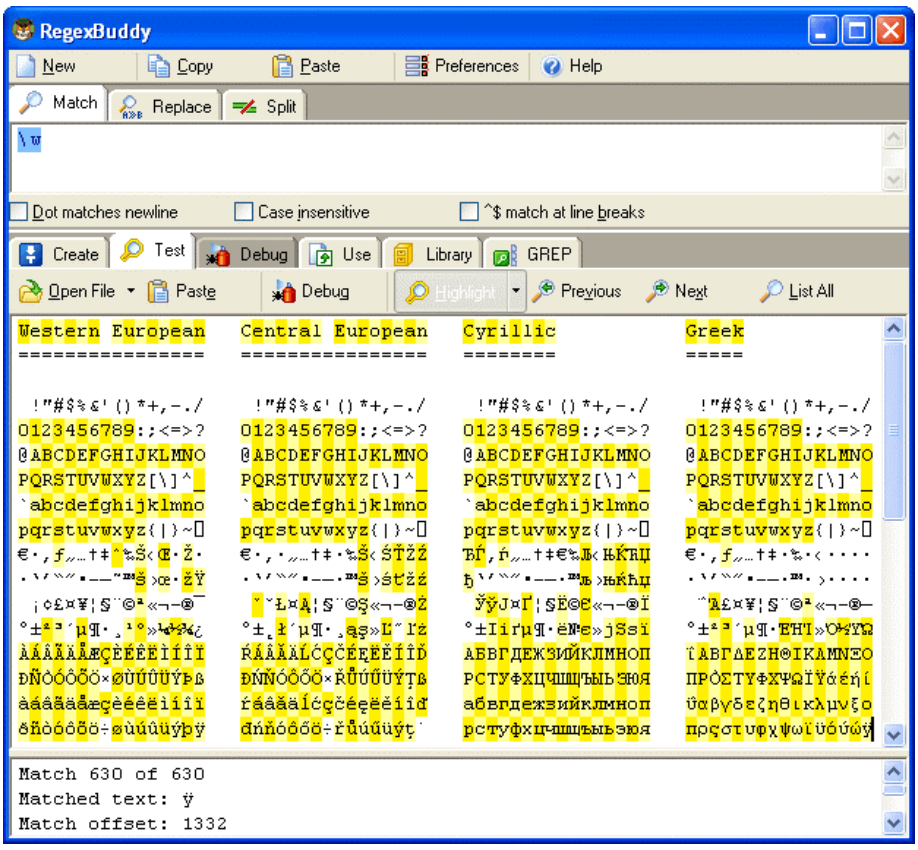
hyphen (-) can be included by escaping them with a backslash, or by placing them in a position where they do not take on their special meaning. I recommend the latter method, since it improves readability. To include a caret, place it anywhere except right after the opening bracket. «`[x^]`» matches an x or a caret. You can put the closing bracket right after the opening bracket, or the negating caret. «`[]x]`» matches a closing bracket or an x. «`[^]x]`» matches any character that is not a closing bracket or an x. The hyphen can be included right after the opening bracket, or right before the closing bracket, or right after the negating caret. Both «`[-x]`» and «`[x-]`» match an x or a hyphen.

You can use non-printable characters in character classes just like you can use them outside of character classes. E.g. «`[$\u20A0]`» matches a dollar or euro sign, assuming your regex flavor supports Unicode.

Shorthand Character Classes

Since certain character classes are used often, a series of shorthand character classes are available. «`\d`» is short for «`[0-9]`».

«`\w`» stands for "word character". Exactly which characters it matches differs between regex flavors. In all flavors, it will include «`[A-Za-z]`». In most, the underscore and digits are also included. In some flavors, word characters from other languages may also match. The best way to find out is to do a couple of tests with the regex flavor you are using. In the screen shot, you can see the characters matched by «`\w`» in RegexBuddy using various scripts.



«`\s`» stands for "whitespace character". Again, which characters this actually includes, depends on the regex flavor. In all flavors discussed in this tutorial, it includes «`[ \t]`». That is: «`\s`» will match a space or a tab. In most flavors, it also includes a carriage return or a line feed as in «`[ \t\r\n]`». Some flavors include additional, rarely used non-printable characters such as vertical tab and form feed.

Shorthand character classes can be used both inside and outside the square brackets. «`\s\d`» matches a whitespace character followed by a digit. «`[\s\d]`» matches a single character that is either whitespace or a digit. When applied to "`1 + 2 = 3`", the former regex will match „ 2" (space two), while the latter matches „1" (one). «`[\da-fA-F]`» matches a hexadecimal digit, and is equivalent to «`[0-9a-fA-F]`».

Negated Shorthand Character Classes

The above three shorthands also have negated versions. «`\D`» is the same as «`[^\d]`», «`\W`» is short for «`[^\w]`» and «`\S`» is the equivalent of «`[^\s]`».

Be careful when using the negated shorthands inside square brackets. «`[\D\S]`» is *not* the same as «`[^\d\s]`». The latter will match any character that is not a digit or whitespace. So it will match „`x`”, but not “`8`”. The former, however, will match any character that is either not a digit, or is not whitespace. Because a digit is not whitespace, and whitespace is not a digit, «`[\D\S]`» will match any character, digit, whitespace or otherwise.

Repeating Character Classes

If you repeat a character class by using the «`?`», «`*`» or «`+`» operators, you will repeat the entire character class, and not just the character that it matched. The regex «`[0-9]+`» can match „`837`” as well as „`222`”.

If you want to repeat the matched character, rather than the class, you will need to use backreferences. «`([0-9])\1+`» will match „`222`” but not “`837`”. When applied to the string “`833337`”, it will match „`3333`” in the middle of this string. If you do not want that, you need to use lookahead and lookbehind.

But I digress. I did not yet explain how character classes work inside the regex engine. Let us take a look at that first.

Looking Inside The Regex Engine

As I already said: the order of the characters inside a character class does not matter. «`gr[ae]y`» will match „`grey`” in “`Is his hair grey or gray?`”, because that is the *leftmost match*. We already saw how the engine applies a regex consisting only of literal characters. Below, I will explain how it applies a regex that has more than one permutation. That is: «`gr[ae]y`» can match both „`gray`” and „`grey`”.

Nothing noteworthy happens for the first twelve characters in the string. The engine will fail to match «`g`» at every step, and continue with the next character in the string. When the engine arrives at the 13th character, „`g`” is matched. The engine will then try to match the remainder of the regex with the text. The next token in the regex is the literal «`r`», which matches the next character in the text. So the third token, «`[ae]`» is attempted at the next character in the text (“`e`”). The character class gives the engine two options: match «`a`» or match «`e`». It will first attempt to match «`a`», and fail.

But because we are using a regex-directed engine, it must continue trying to match all the other permutations of the regex pattern before deciding that the regex cannot be matched with the text starting at character 13. So it will continue with the other option, and find that «`e`» matches „`e`”. The last regex token is «`y`», which can be matched with the following character as well. The engine has found a complete match with the text starting at character 13. It will return „`grey`” as the match result, and look no further. Again, the *leftmost match* was returned, even though we put the «`a`» first in the character class, and „`gray`” could have been matched in the string. But the engine simply did not get that far, because another equally valid match was found to the left of it.

5. The Dot Matches (Almost) Any Character

In regular expressions, the dot or period is one of the most commonly used metacharacters. Unfortunately, it is also the most commonly misused metacharacter.

The dot matches a single character, without caring what that character is. The only exception are newlinecharacters. In all regex flavors discussed in this tutorial, the dot will *not* match a newline character by default. So by default, the dot is short for the negated character class «`[^\n]`» (UNIX regex flavors) or «`[^\r\n]`» (Windows regex flavors).

This exception exists mostly because of historic reasons. The first tools that used regular expressions were line-based. They would read a file line by line, and apply the regular expression separately to each line. The effect is that with these tools, the string could never contain newlines, so the dot could never match them.

Modern tools and languages can apply regular expressions to very large strings or even entire files. All regex flavors discussed here have an option to make the dot match all characters, including newlines. In RegexBuddy, EditPad Pro or PowerGREP, you simply tick the checkbox labeled “dot matches newline”.

In Perl, the mode where the dot also matches newlines is called "single-line mode". This is a bit unfortunate, because it is easy to mix up this term with “multi-line mode”. Multi-line mode only affects anchors, and single-line mode only affects the dot. You can activate single-line mode by adding an s after the regex code, like this: `m/^regex$/s;`.

Other languages and regex libraries have adopted Perl's terminology. When using the regex classes of the .NET framework, you activate this mode by specifying `RegexOptions.Singleline`, such as in `Regex.Match(“string”, “regex”, RegexOptions.Singleline)`.

In all programming languages and regex libraries I know, activating single-line mode has no effect other than making the dot match newlines. So if you expose this option to your users, please give it a clearer label like was done in RegexBuddy, EditPad Pro and PowerGREP.

JavaScript and VBScript do not have an option to make the dot match line break characters. In those languages, you can use a character class such as «`[\s\S]`» to match any character. This character matches a character that is either a whitespace character (including line break characters), or a character that is not a whitespace character. Since all characters are either whitespace or non-whitespace, this character class matches any character.

Use The Dot Sparingly

The dot is a very powerful regex metacharacter. It allows you to be lazy. Put in a dot, and everything will match just fine when you test the regex on valid data. The problem is that the regex will also match in cases where it should not match. If you are new to regular expressions, some of these cases may not be so obvious at first.

I will illustrate this with a simple example. Let's say we want to match a date in mm/dd/yy format, but we want to leave the user the choice of date separators. The quick solution is «`\d\d.\d\d.\d\d`». Seems fine at first. It will match a date like „`02/12/03`” just fine. Trouble is: „`02512703`” is also considered a valid date by

this regular expression. In this match, the first dot matched „5”, and the second matched „7”. Obviously not what we intended.

«\d\d[- /.]\d\d[- /.]\d\d» is a better solution. This regex allows a dash, space, dot and forward slash as date separators. Remember that the dot is not a metacharacter inside a character class, so we do not need to escape it with a backslash.

This regex is still far from perfect. It matches „99/99/99” as a valid date. «[0-1]\d[- /.][0-3]\d[-/.]\d\d» is a step ahead, though it will still match „19/39/99”. How perfect you want your regex to be depends on what you want to do with it. If you are validating user input, it has to be perfect. If you are parsing data files from a known source that generates its files in the same way every time, our last attempt is probably more than sufficient to parse the data without errors. You can find a better regex to match dates in the example section.

Use Negated Character Sets Instead of the Dot

I will explain this in depth when I present you the repeat operators star and plus, but the warning is important enough to mention it here as well. I will illustrate with an example.

Suppose you want to match a double-quoted string. Sounds easy. We can have any number of any character between the double quotes, so «".*"» seems to do the trick just fine. The dot matches any character, and the star allows the dot to be repeated any number of times, including zero. If you test this regex on “Put a “string” between double quotes”, it will match „“string”” just fine. Now go ahead and test it on “Houston, we have a problem with “string one” and “string two”. Please respond.”

Ouch. The regex matches „“string one” and “string two””. Definitely not what we intended. The reason for this is that the star is *greedy*.

In the date-matching example, we improved our regex by replacing the dot with a character class. Here, we will do the same. Our original definition of a double-quoted string was faulty. We do not want any number of *any character* between the quotes. We want any number of characters that are not double quotes or newlines between the quotes. So the proper regex is «"[^"\r\n]*"».

6. Start of String and End of String Anchors

Thus far, I have explained literal characters and character classes. In both cases, putting one in a regex will cause the regex engine to try to match a single character.

Anchors are a different breed. They do not match any character at all. Instead, they match a position before, after or between characters. They can be used to "anchor" the regex match at a certain position. The caret «^» matches the position before the first character in the string. Applying «^a» to "abc" matches „a". «^b» will not match "abc" at all, because the «b» cannot be matched right after the start of the string, matched by «^». See below for the inside view of the regex engine.

Similarly, «$» matches right after the last character in the string. «c$» matches „c" in "abc", while «a$» does not match at all.

Useful Applications

When using regular expressions in a programming language to validate user input, using anchors is very important. If you use the code `if ($input =~ m/\d+/)` in a Perl script to see if the user entered an integer number, it will accept the input even if the user entered "qsdf4ghjk", because «\d+» matches the 4. The correct regex to use is «^\d+$». Because "start of string" must be matched before the match of «\d+», and "end of string" must be matched right after it, the entire string must consist of digits for «^\d+$» to be able to match.

It is easy for the user to accidentally type in a space. When Perl reads from a line from a text file, the line break will also be stored in the variable. So before validating input, it is good practice to trim leading and trailing whitespace. «^\s+» matches leading whitespace and «\s+$» matches trailing whitespace. In Perl, you could use `$input =~ s/^\s+|\s+$//g`. Handy use of alternation and /g allows us to do this in a single line of code.

Using ^ and $ as Start of Line and End of Line Anchors

If you have a string consisting of multiple lines, like "`first line\nsecond line`" (where \n indicates a line break), it is often desirable to work with lines, rather than the entire string. Therefore, all the regex engines discussed in this tutorial have the option to expand the meaning of both anchors. «^» can then match at the start of the string (before the "f" in the above string), as well as after each line break (between "\n" and "s"). Likewise, «$» will still match at the end of the string (after the last "e"), and also before every line break (between "e" and "\n").

In text editors like EditPad Pro or GNU Emacs, and regex tools like PowerGREP, the caret and dollar always match at the start and end of each line. This makes sense because those applications are designed to work with entire files, rather than short strings.

In all programming languages and libraries discussed in this book , except Ruby, you have to explicitly activate this extended functionality. It is traditionally called "multi-line mode". In Perl, you do this by adding an m after the regex code, like this: `m/^regex$/m;`. In .NET, the anchors match before and after newlines when you specify `RegexOptions.Multiline`, such as in `Regex.Match("string", "regex", RegexOptions.Multiline)`.

Permanent Start of String and End of String Anchors

«\A» only ever matches at the start of the string. Likewise, «\Z» only ever matches at the end of the string. These two tokens never match at line breaks. This is true in all regex flavors discussed in this tutorial, even when you turn on "multiline mode". In EditPad Pro and PowerGREP, where the caret and dollar always match at the start and end of lines, «\A» and «\Z» only match at the start and the end of the entire file.

Zero-Length Matches

We saw that the anchors match at a position, rather than matching a character. This means that when a regex only consists of one or more anchors, it can result in a zero-length match. Depending on the situation, this can be very useful or undesirable. Using «^\d*$» to test if the user entered a number (notice the use of the star instead of the plus), would cause the script to accept an empty string as a valid input. See below.

However, matching only a position can be very useful. In email, for example, it is common to prepend a "greater than" symbol and a space to each line of the quoted message. In VB.NET, we can easily do this with `Dim Quoted as String = Regex.Replace(Original, "^", "> ", RegexOptions.Multiline)`. We are using multi-line mode, so the regex «^» matches at the start of the quoted message, and after each newline. The Regex.Replace method will remove the regex match from the string, and insert the replacement string (greater than symbol and a space). Since the match does not include any characters, nothing is deleted. However, the match does include a starting position, and the replacement string is inserted there, just like we want it.

Strings Ending with a Line Break

Even though «\Z» and «$» only match at the end of the string (when the option for the caret and dollar to match at embedded line breaks is off), there is one exception. If the string ends with a line break, then «\Z» and «$» will match at the position before that line break, rather than at the very end of the string. This "enhancement" was introduced by Perl, and is copied by many regex flavors, including Java, .NET and PCRE. In Perl, when reading a line from a file, the resulting string will end with a line break. Reading a line from a file with the text "joe" results in the string "joe\n". When applied to this string, both «^[a-z]+$» and «\A[a-z]+\Z» will match „joe".

If you only want a match at the absolute very end of the string, use «\z» (lower case z instead of upper case Z). «\A[a-z]+\z» does not match "joe\n". «\z» matches after the line break, which is not matched by the character class.

Looking Inside the Regex Engine

Let's see what happens when we try to match «^4$» to "749\n486\n4" (where \n represents a newline character) in multi-line mode. As usual, the regex engine starts at the first character: "7". The first token in the regular expression is «^». Since this token is a zero-width token, the engine does not try to match it with the character, but rather with the position before the character that the regex engine has reached so far. «^» indeed matches the position before "7". The engine then advances to the next regex token: «4». Since the previous token was zero-width, the regex engine does *not* advance to the next character in the string. It remains at "7". «4» is a literal character, which does not match "7". There are no other permutations of the

regex, so the engine starts again with the first regex token, at the next character: “4”. This time, «^» cannot match at the position before the 4. This position is preceded by a character, and that character is not a newline. The engine continues at “9”, and fails again. The next attempt, at “\n”, also fails. Again, the position before “\n” is preceded by a character, “9”, and that character is not a newline.

Then, the regex engine arrives at the second “4” in the string. The «^» can match at the position before the “4”, because it is preceded by a newline character. Again, the regex engine advances to the next regex token, «4», but does not advance the character position in the string. «4» matches „4”, and the engine advances both the regex token and the string character. Now the engine attempts to match «$» at the position before (indeed: before) the “8”. The dollar cannot match here, because this position is followed by a character, and that character is not a newline.

Yet again, the engine must try to match the first token again. Previously, it was successfully matched at the second “4”, so the engine continues at the next character, “8”, where the caret does not match. Same at the six and the newline.

Finally, the regex engine tries to match the first token at the third “4” in the string. With success. After that, the engine successfully matches «4» with „4”. The current regex token is advanced to «$», and the current character is advanced to the very last position in the string: the void after the string. No regex token that needs a character to match can match here. Not even a negated character class. However, we are trying to match a dollar sign, and the mighty dollar is a strange beast. It is zero-width, so it will try to match the position before the current character. It does not matter that this “character” is the void after the string. In fact, the dollar will check the current character. It must be either a newline, or the void after the string, for «$» to match the position before the current character. Since that is the case after the example, the dollar matches successfully. Since «$» was the last token in the regex, the engine has found a successful match: the last „4” in the string.

Another Inside Look

Earlier I mentioned that «^\d*$» would successfully match an empty string. Let's see why. There is only one “character” position in an empty string: the void after the string. The first token in the regex is «^». It matches the position before the void after the string, because it is preceded by the void before the string. The next token is «\d*». As we will see later, one of the star's effects is that it makes the «\d», in this case, optional. The engine will try to match «\d» with the void after the string. That fails, but the star turns the failure of the «\d» into a zero-width success. The engine will proceed with the next regex token, without advancing the position in the string. So the engine arrives at «$», and the void after the string. We already saw that those match. At this point, the entire regex has matched the empty string, and the engine reports success.

Caution for Programmers

A regular expression such as «$» all by itself can indeed match after the string. If you would query the engine for the character position, it would return the length of the string if string indices are zero-based, or the length+1 if string indices are one-based in your programming language. If you would query the engine for the length of the match, it would return zero.

What you have to watch out for is that String[Regex.MatchPosition] may cause an access violation or segmentation fault, because MatchPosition can point to the void after the string. This can also happen with «^» and «^$» if the last character in the string is a newline.

7. Word Boundaries

The metacharacter «\b» is an anchor like the caret and the dollar sign. It matches at a position that is called a “word boundary”. This match is zero-length.

There are four different positions that qualify as word boundaries:

- Before the first character in the string, if the first character is a word character.
- After the last character in the string, if the last character is a word character.
- Between a word character and a non-word character following right after the word character.
- Between a non-word character and a word character following right after the non-word character.

Simply put: «\b» allows you to perform a “whole words only” search using a regular expression in the form of «\bword\b». A “word character” is a character that can be used to form words. All characters that are not “word characters” are “non-word characters”. The exact list of characters is different for each regex flavor, but all word characters are always matched by the short-hand character class «\w». All non-word characters are always matched by «\W».

In Perl and the other regex flavors discussed in this tutorial, there is only one metacharacter that matches both before a word and after a word. This is because any position between characters can never be both at the start and at the end of a word. Using only one operator makes things easier for you.

Note that «\w» usually also matches digits. So «\b4\b» can be used to match a 4 that is not part of a larger number. This regex will not match “`44 sheets of a4`”. So saying "«\b» matches before and after an alphanumeric sequence“ is more exact than saying ”before and after a word".

Negated Word Boundary

«\B» is the negated version of «\b». «\B» matches at every position where «\b» does not. Effectively, «\B» matches at any position between two word characters as well as at any position between two non-word characters.

Looking Inside the Regex Engine

Let's see what happens when we apply the regex «\bis\b» to the string “`This island is beautiful`”. The engine starts with the first token «\b» at the first character “`T`”. Since this token is zero-length, the position before the character is inspected. «\b» matches here, because the T is a word character and the character before it is the void before the start of the string. The engine continues with the next token: the literal «i». The engine does not advance to the next character in the string, because the previous regex token was zero-width. «i» does not match “`T`”, so the engine retries the first token at the next character position.

«\b» cannot match at the position between the “`T`” and the “`h`”. It cannot match between the “`h`” and the “`i`” either, and neither between the “`i`” and the “`s`”.

The next character in the string is a space. «\b» matches here because the space is not a word character, and the preceding character is. Again, the engine continues with the «i» which does not match with the space.

Advancing a character and restarting with the first regex token, «\b» matches between the space and the second “i” in the string. Continuing, the regex engine finds that «i» matches „i” and «s» matches „s”. Now, the engine tries to match the second «\b» at the position before the “l”. This fails because this position is between two word characters. The engine reverts to the start of the regex and advances one character to the “s” in “island”. Again, the «\b» fails to match and continues to do so until the second space is reached. It matches there, but matching the «i» fails.

But «\b» matches at the position before the third “i” in the string. The engine continues, and finds that «i» matches „i” and «s» matches «s». The last token in the regex, «\b», also matches at the position before the second space in the string because the space is not a word character, and the character before it is.

The engine has successfully matched the word „is” in our string, skipping the two earlier occurrences of the characters i and s. If we had used the regular expression «is», it would have matched the „is” in “This”.

8. Alternation with The Vertical Bar or Pipe Symbol

I already explained how you can use character classes to match a single character out of several possible characters. Alternation is similar. You can use alternation to match a single regular expression out of several possible regular expressions. If you want to search for the literal text «`cat`» or «`dog`», separate both options with a vertical bar or pipe symbol: «`cat|dog`». If you want more options, simply expand the list: «`cat|dog|mouse|fish`» .

The alternation operator has the lowest precedence of all regex operators. That is, it tells the regex engine to match either everything to the left of the vertical bar, or everything to the right of the vertical bar. If you want to limit the reach of the alternation, you will need to use round brackets for grouping. If we want to improve the first example to match whole words only, we would need to use «`\b(cat|dog)\b`». This tells the regex engine to find a word boundary, then either “cat” or “dog”, and then another word boundary. If we had omitted the round brackets, the regex engine would have searched for “a word boundary followed by cat”, or, "dog followed by a word boundary.

Remember That The Regex Engine Is Eager

I already explained that the regex engine is eager. It will stop searching as soon as it finds a valid match. The consequence is that in certain situations, the order of the alternatives matters. Suppose you want to use a regex to match a list of function names in a programming language: Get, GetValue, Set or SetValue. The obvious solution is «`Get|GetValue|Set|SetValue`». Let's see how this works out when the string is “`SetValue`”.

The regex engine starts at the first token in the regex, «`G`», and at the first character in the string, “`S`”. The match fails. However, the regex engine studied the entire regular expression before starting. So it knows that this regular expression uses alternation, and that the entire regex has not failed yet. So it continues with the second option, being the second «`G`» in the regex. The match fails again. The next token is the first «`S`» in the regex. The match succeeds, and the engine continues with the next character in the string, as well as the next token in the regex. The next token in the regex is the «`e`» after the «`S`» that just successfully matched. «`e`» matches „`e`”. The next token, «`t`» matches „`t`”.

At this point, the third option in the alternation has been successfully matched. Because the regex engine is eager, it considers the entire alternation to have been successfully matched as soon as one of the options has. In this example, there are no other tokens in the regex outside the alternation, so the entire regex has successfully matched „`Set`” in “`SetValue`”.

Contrary to what we intended, the regex did not match the entire string. There are several solutions. One option is to take into account that the regex engine is eager, and change the order of the options. If we use «`GetValue|Get|SetValue|Set`», «`SetValue`» will be attempted before «`Set`», and the engine will match the entire string. We could also combine the four options into two and use the question mark to make part of them optional: «`Get(Value)?|Set(Value)?`». Because the question mark is greedy, «`SetValue`» will be attempted before «`Set`».

The best option is probably to express the fact that we only want to match complete words. We do not want to match Set or SetValue if the string is “`SetValueFunction`”. So the solution is «`\b(Get|GetValue|Set|SetValue)\b`» or «`\b(Get(Value)?|Set(Value)?)\b`». Since all options have the same end, we can optimize this further to «`\b(Get|Set)(Value)?\b`» .

9. Optional Items

The question mark makes the preceding token in the regular expression optional. E.g.: «`colou?r`» matches both „`colour`” and „`color`”.

You can make several tokens optional by grouping them together using round brackets, and placing the question mark after the closing bracket. E.g.: «`Nov(ember)?`» will match „`Nov`” and „`November`”.

You can write a regular expression that matches many alternatives by including more than one question mark. «`Feb(ruary)? 23(rd)?`» matches „`February 23rd`”, „`February 23`”, „`Feb 23rd`” and „`Feb 23`”.

Important Regex Concept: Greediness

With the question mark, I have introduced the first metacharacter that is *greedy*. The question mark gives the regex engine two choices: try to match the part the question mark applies to, or do not try to match it. The engine will always try to match that part. Only if this causes the entire regular expression to fail, will the engine try ignoring the part the question mark applies to.

The effect is that if you apply the regex «`Feb 23(rd)?`» to the string “`Today is Feb 23rd, 2003`”, the match will always be „`Feb 23rd`” and not „`Feb 23`”. You can make the question mark *lazy* (i.e. turn off the greediness) by putting a second question mark after the first.

I will say a lot more about greediness when discussing the other repetition operators.

Looking Inside The Regex Engine

Let's apply the regular expression «`colou?r`» to the string “`The colonel likes the color green`”.

The first token in the regex is the literal «`c`». The first position where it matches successfully is the „`c`” in “`colonel`”. The engine continues, and finds that «`o`» matches „`o`”, «`l`» matches „`l`” and another «`o`» matches „`o`”. Then the engine checks whether «`u`» matches “`n`”. This fails. However, the question mark tells the regex engine that failing to match «`u`» is acceptable. Therefore, the engine will skip ahead to the next regex token: «`r`». But this fails to match “`n`” as well. Now, the engine can only conclude that the entire regular expression cannot be matched starting at the „`c`” in “`colonel`”. Therefore, the engine starts again trying to match «`c`» to the first `o` in “`colonel`”.

After a series of failures, «`c`» will match with the „`c`” in “`color`”, and «`o`», «`l`» and «`o`» match the following characters. Now the engine checks whether «`u`» matches “`r`”. This fails. Again: no problem. The question mark allows the engine to continue with «`r`». This matches „`r`” and the engine reports that the regex successfully matched „`color`” in our string.

10. Repetition with Star and Plus

I already introduced one repetition operator or quantifier: the question mark. It tells the engine to attempt match the preceding token zero times or once, in effect making it optional.

The asterisk or star tells the engine to attempt to match the preceding token zero or more times. The plus tells the engine to attempt to match the preceding token once or more. «`<[A-Za-z][A-Za-z0-9]*>`» matches an HTML tag without any attributes. The sharp brackets are literals. The first character class matches a letter. The second character class matches a letter or digit. The star repeats the second character class. Because we used the star, it's OK if the second character class matches nothing. So our regex will match a tag like „`<B>`". When matching „`<HTML>`", the first character class will match „`H`". The star will cause the second character class to be repeated three times, matching „`T`", „`M`" and „`L`" with each step.

I could also have used «`<[A-Za-z0-9]+>`». I did not, because this regex would match „`<1>`", which is not a valid HTML tag. But this regex may be sufficient if you know the string you are searching through does not contain any such invalid tags.

Limiting Repetition

Modern regex flavors, like those discussed in this tutorial, have an additional repetition operator that allows you to specify how many times a token can be repeated. The syntax is `{`*`min`*`,`*`max`*`}`, where *min* is a positive integer number indicating the minimum number of matches, and *max* is an integer equal to or greater than *min* indicating the maximum number of matches. If the comma is present but *max* is omitted, the maximum number of matches is infinite. So «`{0,}`» is the same as «`*`», and «`{1,}`» is the same as «`+`». Omitting both the comma and *max* tells the engine to repeat the token exactly *min* times.

You could use «`\b[1-9][0-9]{3}\b`» to match a number between 1000 and 9999. «`\b[1-9][0-9]{2,4}\b`» matches a number between 100 and 99999. Notice the use of the word boundaries.

Watch Out for The Greediness!

Suppose you want to use a regex to match an HTML tag. You know that the input will be a valid HTML file, so the regular expression does not need to exclude any invalid use of sharp brackets. If it sits between sharp brackets, it is an HTML tag.

Most people new to regular expressions will attempt to use «`<.+>`». They will be surprised when they test it on a string like "`This is a <EM>first</EM> test`". You might expect the regex to match „`<EM>`" and when continuing after that match, „`</EM>`".

But it does not. The regex will match „`<EM>first</EM>`". Obviously not what we wanted. The reason is that the plus is *greedy*. That is, the plus causes the regex engine to repeat the preceding token as often as possible. Only if that causes the entire regex to fail, will the regex engine *backtrack*. That is, it will go back to the plus, make it give up the last iteration, and proceed with the remainder of the regex. Let's take a look inside the regex engine to see in detail how this works and why this causes our regex to fail. After that, I will present you with two possible solutions.

Like the plus, the star and the repetition using curly braces are greedy.

Looking Inside The Regex Engine

The first token in the regex is «<». This is a literal. As we already know, the first place where it will match is the first „<" in the string. The next token is the dot, which matches any character except newlines. The dot is repeated by the plus. The plus is *greedy*. Therefore, the engine will repeat the dot as many times as it can. The dot matches „E", so the regex continues to try to match the dot with the next character. „M" is matched, and the dot is repeated once more. The next character is the ">". You should see the problem by now. The dot matches the „>", and the engine continues repeating the dot. The dot will match all remaining characters in the string. The dot fails when the engine has reached the void after the end of the string. Only at this point does the regex engine continue with the next token: «>».

So far, «<.+» has matched „<EM>first</EM> test" and the engine has arrived at the end of the string. «>» cannot match here. The engine remembers that the plus has repeated the dot more often than is required. (Remember that the plus *requires* the dot to match only once.) Rather than admitting failure, the engine will *backtrack*. It will reduce the repetition of the plus by one, and then continue trying the remainder of the regex.

So the match of «.+» is reduced to „EM>first</EM> tes". The next token in the regex is still «>». But now the next character in the string is the last "t". Again, these cannot match, causing the engine to backtrack further. The total match so far is reduced to „<EM>first</EM> te". But «>» still cannot match. So the engine continues backtracking until the match of «.+» is reduced to „EM>first</EM". Now, «>» can match the next character in the string. The last token in the regex has been matched. The engine reports that „<EM>first</EM>" has been successfully matched.

Remember that the regex engine is *eager* to return a match. It will not continue backtracking further to see if there is another possible match. It will report the first valid match it finds. Because of greediness, this is the leftmost longest match.

Laziness Instead of Greediness

The quick fix to this problem is to make the plus lazy instead of greedy. You can do that by putting a question markbehind the plus in the regex. You can do the same with the star, the curly braces and the question mark itself. So our example becomes «<.+?>». Let's have another look inside the regex engine.

Again, «<» matches the first „<" in the string. The next token is the dot, this time repeated by a lazy plus. This tells the regex engine to repeat the dot as few times as possible. The minimum is one. So the engine matches the dot with „E". The requirement has been met, and the engine continues with «>» and "M". This fails. Again, the engine will *backtrack*. But this time, the backtracking will force the lazy plus to expand rather than reduce its reach. So the match of «.+» is expanded to „EM", and the engine tries again to continue with «>». Now, „>" is matched successfully. The last token in the regex has been matched. The engine reports that „<EM>" has been successfully matched. That's more like it.

An Alternative to Laziness

In this case, there is a better option than making the plus lazy. We can use a greedy plus and a negated character class: «<[^>]+>». The reason why this is better is because of the backtracking. When using the lazy plus, the engine has to backtrack for each character in the HTML tag that it is trying to match. When using the negated character class, no backtracking occurs at all when the string contains valid HTML code.

Backtracking slows down the regex engine. You will not notice the difference when doing a single search in a text editor. But you will save plenty of CPU cycles when using such a regex is used repeatedly in a tight loop in a script that you are writing, or perhaps in a custom syntax coloring scheme for EditPad Pro.

Finally, remember that this tutorial only talks about regex-directed engines. Text-directed engines do not backtrack. They do not get the speed penalty, but they also do not support lazy repetition operators.

11. Use Round Brackets for Grouping

By placing part of a regular expression inside round brackets or parentheses, you can group that part of the regular expression together. This allows you to apply a regex operator, e.g. a repetition operator, to the entire group. I have already used round brackets for this purpose in previous topics throughout this tutorial.

Note that only round brackets can be used for grouping. Square brackets define a character class, and curly braces are used by a special repetition operator.

Round Brackets Create a Backreference

Besides grouping part of a regular expression together, round brackets also create a "backreference". A backreference stores the part of the string matched by the part of the regular expression inside the parentheses.

That is, unless you use non-capturing parentheses. Remembering part of the regex match in a backreference, slows down the regex engine because it has more work to do. If you do not use the backreference, you can speed things up by using non-capturing parentheses, at the expense of making your regular expression slightly harder to read.

The regex «`Set(Value)?`» matches „`Set`" or „`SetValue`". In the first case, the first backreference will be empty, because it did not match anything. In the second case, the first backreference will contain „`Value`".

If you do not use the backreference, you can optimize this regular expression into «`Set(?:Value)?`». The question mark and the colon after the opening round bracket are the special syntax that you can use to tell the regex engine that this pair of brackets should not create a backreference. Note the question mark after the opening bracket is unrelated to the question mark at the end of the regex. That question mark is the regex operator that makes the previous token optional. This operator cannot appear after an opening round bracket, because an opening bracket by itself is not a valid regex token. Therefore, there is no confusion between the question mark as an operator to make a token optional, and the question mark as a character to change the properties of a pair of round brackets. The colon indicates that the change we want to make is to turn off capturing the backreference.

How to Use Backreferences

Backreferences allow you to reuse part of the regex match. You can reuse it inside the regular expression (see below), or afterwards. What you can do with it afterwards, depends on the tool you are using. In EditPad Pro or PowerGREP, you can use the backreference in the replacement text during a search-and-replace operation by typing `\1` (backslash one) into the replacement text. If you searched for «`EditPad (Lite|Pro)`» and use "`\1 version`" as the replacement, the actual replacement will be "`Lite version`" in case „`EditPad Lite`" was matched, and "`Pro version`" in case „`EditPad Pro`" was matched.

EditPad Pro and PowerGREP have a unique feature that allows you to change the case of the backreference. `\U1` inserts the first backreference in uppercase, `\L1` in lowercase and `\F1` with the first character in uppercase and the remainder in lowercase. Finally, `\I1` inserts it with the first letter of each word capitalized, and the other letters in lowercase.

Regex libraries in programming languages also provide access to the backreference. In Perl, you can use the magic variables `$1`, `$2`, etc. to access the part of the string matched by the backreference. In .NET (dot net), you can use the `Match` object that is returned by the `Match` method of the `Regex` class. This object has a property called `Groups`, which is a collection of Group objects. To get the string matched by the third backreference in C#, you can use `MyMatch.Groups[3].Value`.

The .NET (dot net) Regex class also has a method `Replace` that can do a regex-based search-and-replace on a string. In the replacement text, you can use `$1`, `$2`, etc. to insert backreferences.

To figure out the number of a particular backreference, scan the regular expression from left to right and count the opening round brackets. The first bracket starts backreference number one, the second number two, etc. Non-capturing parentheses are not counted. This fact means that non-capturing parentheses have another benefit: you can insert them into a regular expression without changing the numbers assigned to the backreferences. This can be very useful when modifying a complex regular expression.

The Entire Regex Match As Backreference Zero

Certain tools make the entire regex match available as backreference zero. In EditPad Pro or PowerGREP, you can use the entire regex match in the replacement text during a search and replace operation by typing `\0` (backslash zero) into the replacement text. In Perl, the magic variable `$&` holds the entire regex match. Libraries like .NET (dot net) where backreferences are made available as an array or numbered list, the item with index zero holds the entire regex match. Using backreference zero is more efficient than putting an extra pair of round brackets around the entire regex, because that would force the engine to continuously keep an extra copy of the entire regex match.

Using Backreferences in The Regular Expression

Backreferences can not only be used after a match has been found, but also during the match. Suppose you want to match a pair of opening and closing HTML tags, and the text in between. By putting the opening tag into a backreference, we can reuse the name of the tag for the closing tag. Here's how: «`<([A-Z][A-Z0-9]*)[^>]*>.*?</\1>`» . This regex contains only one pair of parentheses, which capture the string matched by «`[A-Z][A-Z0-9]*`» into the first backreference. This backreference is reused with «`\1`» (backslash one). The «`/`» before it is simply the forward slash in the closing HTML tag that we are trying to match.

You can reuse the same backreference more than once. «`([a-c])x\1x\1`» will match „`axaxa`”, „`bxbxb`” and „`cxcxc`”. If a backreference was not used in a particular match attempt (such as in the first example where the question mark made the first backreference optional), it is simply empty. Using an empty backreference in the regex is perfectly fine. It will simply be replaced with nothingness.

A backreference cannot be used inside itself. «`([abc]\1)`» will not work. Depending on your regex flavor, it will either give an error message, or it will fail to match anything without an error message. Therefore, `\0` cannot be used inside a regex, only in the replacement.

Looking Inside The Regex Engine

Let's see how the regex engine applies the above regex to the string "`Testing <B><I>bold italic</I></B> text`". The first token in the regex is the literal «<». The regex engine will traverse the string until it can match at the first „<" in the string. The next token is «`[A-Z]`». The regex engine also takes note that it is now inside the first pair of capturing parentheses. «`[A-Z]`» matches „`B`". The engine advances to «`[A-Z0-9]`» and "`>`". This match fails. However, because of the star, that's perfectly fine. The position in the string remains at "`>`". The position in the regex is advanced to «`[^>]`».

This step crosses the closing bracket of the first pair of capturing parentheses. This prompts the regex engine to store what was matched inside them into the first backreference. In this case, „`B`" is stored.

After storing the backreference, the engine proceeds with the match attempt. «`[^>]`» does not match „>". Again, because of another star, this is not a problem. The position in the string remains at "`>`", and position in the regex is advanced to «>». These obviously match. The next token is a dot, repeated by a lazy star. Because of the laziness, the regex engine will initially skip this token, taking note that it should backtrack in case the remainder of the regex fails.

The engine has now arrived at the second «<» in the regex, and the second "<" in the string. These match. The next token is «/». This does not match "`I`", and the engine is forced to backtrack to the dot. The dot matches the second „<" in the string. The star is still lazy, so the engine again takes note of the available backtracking position and advances to «<» and "`I`". These do not match, so the engine again backtracks.

The backtracking continues until the dot has consumed „`<I>bold italic`". At this point, «<» matches the third „<" in the string, and the next token is «/» which matches "/". The next token is «`\1`». Note that the token the backreference, and not «`B`». The engine does not substitute the backreference in the regular expression. Every time the engine arrives at the backreference, it will read the value that was stored. This means that if the engine had backtracked beyond the first pair of capturing parentheses before arriving the second time at «`\1`», the new value stored in the first backreference would be used. But this did not happen here, so „`B`" it is. This fails to match at "`I`", so the engine backtracks again, and the dot consumes the third "<" in the string.

Backtracking continues again until the dot has consumed „`<I>bold italic</I>`". At this point, «<» matches „<" and «/» matches „/". The engine arrives again at «`\1`». The backreference still holds „`B`". «`B`» matches „`B`". The last token in the regex, «>» matches „>". A complete match has been found: „`<B><I>bold italic</I></B>`".

Repetition and Backreferences

As I mentioned in the above inside look, the regex engine does not permanently substitute backreferences in the regular expression. It will use the last match saved into the backreference each time it needs to be used. If a new match is found by capturing parentheses, the previously saved match is overwritten. There is a clear difference between «`([abc]+)`» and «`([abc])+`». Though both successfully match „`cab`", the first regex will put „`cab`" into the first backreference, while the second regex will only store „`b`". That is because in the second regex, the plus caused the pair of parentheses to repeat three times. The first time, „`c`" was stored. The second time „`a`" and the third time „`b`". Each time, the previous value was overwritten, so „`b`" remains.

This also means that «`([abc]+)=\1`» will match „`cab=cab`", and that «`([abc])+=\1`» will not. The reason is that when the engine arrives at «`\1`», it holds «b» which fails to match "`c`". Obvious when you look at a

simple example like this one, but a common cause of difficulty with regular expressions nonetheless. When using backreferences, always double check that you are really capturing what you want.

Useful Example: Checking for Doubled Words

When editing text, doubled words such as “the the” easily creep in. Using the regex «`\b(\w+)\s+\1\b`» in your text editor, you can easily find them. To delete the second word, simply type in “`\1`” as the replacement text and click the Replace button.

Parentheses and Backreferences Cannot Be Used Inside Character Classes

Round brackets cannot be used inside character classes, at least not as metacharacters. When you put a round bracket in a character class, it is treated as a literal character. So the regex «`[(a)b]`» matches „`a`”, „`b`”, „`(`” and „`)`”.

Backreferences also cannot be used inside a character class. The `\1` in regex like «`(a)[\1b]`» will be interpreted as an octal escape in most regex flavors. So this regex will match an „`a`” followed by either «`\x01`» or a «`b`».

12. Named Capturing Groups

All modern regular expression engines support capturing groups, which are numbered from left to right, starting with one. The numbers can then be used in backreferences to match the same text again in the regular expression, or to use part of the regex match for further processing. In a complex regular expression with many capturing groups, the numbering can get a little confusing.

Named Capture with Python, PCRE and PHP

Python's regex module was the first to offer a solution: named capture. By assigning a name to a capturing group, you can easily reference it by name. «`(?P<name>group)`» captures the match of «`group`» into the backreference "name". You can reference the contents of the group with the numbered backreference «`\1`» or the named backreference «`(?P=name)`».

The open source PCRE library has followed Python's example, and offers named capture using the same syntax. The PHP preg functions offer the same functionality, since they are based on PCRE.

Python's `sub()` function allows you to reference a named group as "`\1`" or "`\g<name>`". This does *not* work in PHP. In PHP, you can use double-quoted string interpolation with the `$regs` parameter you passed to `pcre_match()`: "`$regs['name']`".

Named Capture with .NET's System.Text.RegularExpressions

The regular expression classes of the .NET framework also support named capture. Unfortunately, the Microsoft developers decided to invent their own syntax, rather than follow the one pioneered by Python. Currently, no other regex flavor supports Microsoft's version of named capture.

Here is an example with two capturing groups in .NET style: «`(?<first>group)(?'second'group)`». As you can see, .NET offers two syntaxes to create a capturing group: one using sharp brackets, and the other using single quotes. The first syntax is preferable in strings, where single quotes may need to be escaped. The second syntax is preferable in ASP code, where the sharp brackets are used for HTML tags. You can use the pointy bracket flavor and the quoted flavors interchangeably.

To reference a capturing group inside the regex, use «`\k<name>`» or «`\k'name'`». Again, you can use the two syntactic variations interchangeably.

When doing a search-and-replace, you can reference the named group with the familiar dollar sign syntax: "`${name}`". Simply use a name instead of a number between the curly braces.

Names and Numbers for Capturing Groups

Here is where things get a bit ugly. Python and PCRE treat named capturing groups just like unnamed capturing groups, and number both kinds from left to right, starting with one. The regex «`(a)(?P<x>b)(c)(?P<y>d)`» matches „`abcd`" as expected. If you do a search-and-replace with this regex

and the replacement "`\1\2\3\4`", you will get "`abcd`". All four groups were numbered from left to right, from one till four. Easy and logical.

Things are quite a bit more complicated with the .NET framework. The regex «`(a)(?<x>b)(c)(?<y>d)`» again matches „`abcd`". However, if you do a search-and-replace with "`$1$2$3$4`" as the replacement, you will get "`acbd`". Probably not what you expected.

The .NET framework *does* number named capturing groups from left to right, but numbers them *after* all the unnamed groups have been numbered. So the unnamed groups «`(a)`» and «`(c)`» get numbered first, from left to right, starting at one. Then the named groups «`(?<x>b)`» and «`(?<y>d)`» get their numbers, continuing from the unnamed groups, in this case: three.

To make things simple, when using .NET's regex support, just assume that named groups do not get numbered at all, and reference them by name exclusively. To keep things compatible across regex flavors, I strongly recommend that you do not mix named and unnamed capturing groups at all. Either give a group a name, or make it non-capturing as in «`(?:nocapture)`». Non-capturing groups are more efficient, since the regex engine does not need to keep track of their matches.

Other Regex Flavors

EditPad Pro and PowerGREP support both the Python syntax and the .NET syntax for named capture. However, they will number named groups along with unnamed capturing groups, just like Python does.

RegexBuddy also supports both Python's and Microsoft's style. RegexBuddy will convert one flavor of named capture into the other when generating source code snippets for Python, PHP/preg, PHP, or one of the .NET languages.

None of the other regex flavors discussed in this book support named capture.

13. Unicode Regular Expressions

Unicode is a character set that aims to define all characters and glyphs from all human languages, living and dead. With more and more software being required to support multiple languages, or even just *any* language, Unicode has been strongly gaining popularity in recent years. Using different character sets for different languages is simply too cumbersome for programmers and users.

Unfortunately, Unicode brings its own requirements and pitfalls when it comes to regular expressions. Of the regex flavors discussed in this tutorial, Java and the .NET framework use Unicode-based regex engines. Perl supports Unicode starting with version 5.6.

RegexBuddy's regex engine is fully Unicode-based starting with version 2.0.0. RegexBuddy 1.x.x did not support Unicode at all. PowerGREP uses the same Unicode regex engine starting with version 3.0.0. Earlier versions would convert Unicode files to ANSI prior to grepping with an 8-bit (i.e. non-Unicode) regex engine.

Characters, Code Points and Graphemes or How Unicode Makes a Mess

Most people would consider “à” a single character. Unfortunately, it need not be depending on the meaning of the word “character”.

All regex engines discussed in this tutorial treat any single Unicode *code point* as a single character. When this tutorial tells you that the dot matches any single character, this translates into Unicode parlance as “the dot matches any single Unicode code point”. In Unicode, “à” can be encoded as two code points: U+0061 (a) followed by U+0300 (grave accent). In this situation, «`.`» applied to “à” will match „a” without the accent. «`^.$`» will fail to match, since the string consists of two code points. «`^..$`» matches „à”.

The Unicode code point U+0300 (grave accent) is a *combining mark*. Any code point that is not a combining mark can be followed by any number of combining marks. This sequence, like U+0061 U+0300 above, is displayed as a single *grapheme* on the screen.

Unfortunately, “à” can also be encoded with the single Unicode code point U+00E0 (a with grave accent). The reason for this duality is that many historical character sets encode “a with grave accent” as a single character. Unicode's designers thought it would be useful to have a one-on-one mapping with popular legacy character sets, in addition to the Unicode way of separating marks and base letters (which makes arbitrary combinations not supported by legacy character sets possible).

How to Match a Single Unicode Grapheme

Matching a single grapheme, whether it's encoded as a single code point, or as multiple code points using combining marks, is easy in Perl, RegexBuddy and PowerGREP: simply use «`\X`». You can consider «`\X`» the Unicode version of the dot in regex engines that use plain ASCII. There is one difference, though: «`\X`» always matches line break characters, whereas the dot does not match line break characters unless you enable the dot matches newline matching mode.

Java and .NET unfortunately do not support «`\X`» (yet). Use «`\P{M}\p{M}*`» as a substitute. To match any number of graphemes, use «`(?:\P{M}\p{M}*)+`» instead of «`\X+`».

Unicode Character Properties

In addition to complications, Unicode also brings new possibilities. One is that each Unicode character belongs to a certain category. You can match a single character belonging to a particular category with «`\p{}`». You can match a single character *not* belonging to a particular category with «`\P{}`».

Again, "character" really means "Unicode code point". «`\p{L}`» matches a single code point in the category "letter". If your input string is "à" encoded as U+0061 U+0300, it matches „a" without the accent. If the input is "à" encoded as U+00E0, it matches „à" with the accent. The reason is that both the code points U+0061 (a) and U+00E0 (à) are in the category "letter", while U+0300 is in the category "mark".

You should now understand why «`\P{M}\p{M}*`» is the equivalent of «`\X`». «`\P{M}`» matches a code point that is not a combining mark, while «`\p{M}*`» matches zero or more code points that are combining marks. To match a letter including any diacritics, use «`\p{L}\p{M}*`». This last regex will always match „à", regardless of how it is encoded.

In addition to the standard notation, «`\p{L}`», Java, Perl, RegexBuddy and PowerGREP allow you to use the shorthand «`\pL`». In addition to that, Perl, RegexBuddy and PowerGREP also support the longhand «`\p{Letter}`».

- «`\p{L}`» or «`\p{Letter}`»: any kind of letter from any language.
  - «`\p{Ll}`» or «`\p{Lowercase_Letter}`»: a lowercase letter that has an uppercase variant.
  - «`\p{Lu}`» or «`\p{Uppercase_Letter}`»: an uppercase letter that has a lowercase variant.
  - «`\p{Lt}`» or «`\p{Titlecase_Letter}`»: a letter that appears at the start of a word when only the first letter of the word is capitalized.
  - «`\p{L&}`» or «`\p{Letter&}`»: a letter that exists in lowercase and uppercase variants (combination of Ll, Lu and Lt).
  - «`\p{Lm}`» or «`\p{Modifier_Letter}`»: a special character that is used like a letter.
  - «`\p{Lo}`» or «`\p{Other_Letter}`»: a letter or ideograph that does not have lowercase and uppercase variants.
- «`\p{M}`» or «`\p{Mark}`»: a character intended to be combined with another character (e.g. accents, umlauts, enclosing boxes, etc.).
  - «`\p{Mn}`» or «`\p{Non_Spacing_Mark}`»: a character intended to be combined with another character that does not take up extra space (e.g. accents, umlauts, etc.).
  - «`\p{Mc}`» or «`\p{Spacing_Combining_Mark}`»: a character intended to be combined with another character that takes up extra space (vowel signs in many Eastern languages).
  - «`\p{Me}`» or «`\p{Enclosing_Mark}`»: a character that encloses the character is is combined with (circle, square, keycap, etc.).
- «`\p{Z}`» or «`\p{Separator}`»: any kind of whitespace or invisible separator.
  - «`\p{Zs}`» or «`\p{Space_Separator}`»: a whitespace character that is invisible, but does take up space.
  - «`\p{Zl}`» or «`\p{Line_Separator}`»: line separator character U+2028.
  - «`\p{Zp}`» or «`\p{Paragraph_Separator}`»: paragraph separator character U+2029.
- «`\p{S}`» or «`\p{Symbol}`»: math symbols, currency signs, dingbats, box-drawing characters, etc..
  - «`\p{Sm}`» or «`\p{Math_Symbol}`»: any mathematical symbol.
  - «`\p{Sc}`» or «`\p{Currency_Symbol}`»: any currency sign.
  - «`\p{Sk}`» or «`\p{Modifier_Symbol}`»: a combining character (mark) as a full character on its own.
  - «`\p{So}`» or «`\p{Other_Symbol}`»: various symbols that are not math symbols, currency signs, or combining characters.

- «\p{N}» or «\p{Number}»: any kind of numeric character in any script.
  - «\p{Nd}» or «\p{Decimal_Digit_Number}»: a digit zero through nine in any script except ideographic scripts.
  - «\p{Nl}» or «\p{Letter_Number}»: a number that looks like a letter, such as a Roman numeral.
  - «\p{No}» or «\p{Other_Number}»: a superscript or subscript digit, or a number that is not a digit 0..9 (excluding numbers from ideographic scripts).
- «\p{P}» or «\p{Punctuation}»: any kind of punctuation character.
  - «\p{Pd}» or «\p{Dash_Punctuation}»: any kind of hyphen or dash.
  - «\p{Ps}» or «\p{Open_Punctuation}»: any kind of opening bracket.
  - «\p{Pe}» or «\p{Close_Punctuation}»: any kind of closing bracket.
  - «\p{Pi}» or «\p{Initial_Punctuation}»: any kind of opening quote.
  - «\p{Pf}» or «\p{Final_Punctuation}»: any kind of closing quote.
  - «\p{Pc}» or «\p{Connector_Punctuation}»: a punctuation character such as an underscore that connects words.
  - «\p{Po}» or «\p{Other_Punctuation}»: any kind of punctuation character that is not a dash, bracket, quote or connector.
- «\p{C}» or «\p{Other}»: invisible control characters and unused code points.
  - «\p{Cc}» or «\p{Control}»: an ASCII 0x00..0x1F or Latin-1 0x80..0x9F control character.
  - «\p{Cf}» or «\p{Format}»: invisible formatting indicator.
  - «\p{Co}» or «\p{Private_Use}»: any code point reserved for private use.
  - «\p{Cs}» or «\p{Surrogate}»: one half of a surrogate pair in UTF-16 encoding.
  - «\p{Cn}» or «\p{Unassigned}»: any code point to which no character has been assigned.

Do You Need To Worry About Different Encodings?

While you should always keep in mind the pitfalls created by the different ways in which accented characters can be encoded, you don't always have to worry about them. If you know that your input string and your regex use the same style, then you don't have to worry about it at all. This process is called Unicode *normalization*. All programming languages with native Unicode support, such as Java, C# and VB.NET, have library routines for normalizing strings. If you normalize both the subject and regex before attempting the match, there won't be any inconsistencies.

If you are using Java, you can pass the CANON_EQ flag as the second parameter to Pattern.compile(). This tells the Java regex engine to consider *canonically equivalent* characters as identical. E.g. the regex «à» encoded as U+00E0 will match „à” encoded as U+0061 U+0300, and vice versa. None of the other regex engines currently support canonical equivalence while matching.

If you type the à key on the keyboard, all word processors that I know of will insert the code point U+00E0 into the file. So if you're working with text that you typed in yourself, any regex that you type in yourself will match in the same way.

Finally, if you're using PowerGREP to search through text files encoded using a traditional Windows (often called “ANSI”) or ISO-8859 code page, PowerGREP will always use the one-on-one substitution. Since all the Windows or ISO-8859 code pages encode accented characters as a single code point, all software that I know of will use a single Unicode code point for each character when converting the file to Unicode.

Matching a Specific Code Point

To match a specific Unicode code point, use «`\uFFFF`» where FFFF is the hexadecimal number of the code point you want to match. E.g. «`\u00E0`» matches „à”, but only when encoded as a single code point U+00E0.

In Java, the regex token «`\uFFFF`» only matches the specified code point, even when you turned on canonical equivalence. However, the same syntax `\uFFFF` is also used to insert Unicode characters into literal strings in the Java source code. `Pattern.compile("\u00E0")` will match both the single-code-point and double-code-point encodings of „à”, while `Pattern.compile("\\u00E0")` matches only the single-code-point version. Remember that when writing a regex as a Java string literal, backslashes must be escaped. The former Java code compiles the regex «à», while the latter compiles «`\u00E0`». Depending on what you're doing, the difference may be significant.

14. Regex Matching Modes

Most regular expression engines discussed in this tutorial support the following three matching modes:

- `/i` makes the regex match case insensitive.
- `/s` enables "single-line mode". In this mode, the dot matches newlines.
- `/m` enables "multi-line mode". In this mode, the caret and dollar match before and after newlines in the subject string.

Two languages that don't support all of the above three are JavaScript and Ruby. Many regex flavors also have additional modes or options that have single letter equivalents. These differ widely.

Most tools that support regular expressions have checkboxes or similar controls that you can use to turn these modes on or off. Most programming languages allow you to pass option flags when constructing the regex object. E.g. in Perl, `m/regex/i` turns on case insensitivity, while `Pattern.compile("regex", Pattern.CASE_INSENSITIVE)` does the same in Java.

Specifying Modes Inside The Regular Expression

Sometimes, the tool or language does not provide the ability to specify matching options. E.g. the handy `String.matches()` method in Java does not take a parameter for matching options like `Pattern.compile()` does. In that situation, you can add a mode modifier to the start of the regex. E.g. `(?i)` turns on case insensitivity, while `(?ism)` turns on all three options.

Turning Modes On and Off for Only Part of The Regular Expression

Modern regex flavors allow you to apply modifiers to only part of the regular expression. If you insert the modifier `(?ism)` in the middle of the regex, the modifier only applies to the part of the regex to the right of the modifier. You can turn off modes by preceding them with a minus sign. All modes after the minus sign will be turned off. E.g. `(?i-sm)` turns on case insensitivity, and turns off both single-line mode and multi-line mode.

Not all regex flavors support this. The latest versions of all tools and languages discussed in this book do. Older regex flavors usually apply the option to the entire regular expression, no matter where you placed it. You can quickly test this. The regex «`(?i)te(?-i)st`» should match „`test`" and „`TEst`", but not "`teST`" or "`TEST`".

Modifier Spans

Instead of using two modifiers, one to turn an option on, and one to turn it off, you use a modifier span. «`(?i)ignorecase(?-i)casesensitive(?i)ignorecase`» is equivalent to «`(?i)ignorecase(?-i:casesensitive)ignorecase`». You have probably noticed the resemblance between the modifier span and the non-capturing group «`(?:group)`». Technically, the non-capturing group is a modifier span that does not change any modifiers. It is obvious that the modifier span does not create a backreference.

15. Atomic Grouping and Possessive Quantifiers

When discussing the repetition operators or quantifiers, I explained the difference between greedy and lazy repetition. Greediness and laziness determine the order in which the regex engine tries the possible permutations of the regex pattern. A greedy quantifier will first try to repeat the token as many times as possible, and gradually give up matches as the engine backtracks to find an overall match. A lazy quantifier will first repeat the token as few times as required, and gradually expand the match as the engine backtracks through the regex to find an overall match.

Because greediness and laziness change the order in which permutations are tried, they can change the overall regex match. However, they do not change the fact that the regex engine will backtrack to try all possible permutations of the regular expression in case no match can be found. First, let's see why backtracking can lead to problems.

Catastrophic Backtracking

Recently I got a complaint from a customer that EditPad Pro hung (i.e. it stopped responding) when trying to find lines in a comma-delimited text file where the 12th item on a line started with a "`P`". The customer was using the regexp «`^(.*?,){11}P`» .

At first sight, this regex looks like it should do the job just fine. The lazy dot and comma match a single comma-delimited field, and the {11} skips the first 11 fields. Finally, the P checks if the 12th field indeed starts with P. In fact, this is exactly what will happen when the 12th field indeed starts with a P.

The problem rears its ugly head when the 12th field does not start with a P. Let's say the string is "`1,2,3,4,5,6,7,8,9,10,11,12,13`". At that point, the regex engine will backtrack. It will backtrack to the point where «`^(.*?,){11}`» had consumed „`1,2,3,4,5,6,7,8,9,10,11`", giving up the last match of the comma. The next token is again the dot. The dot matches a comma. *The dot matches the comma!* However, the comma does not match the "`1`" in the 12th field, so the dot continues until the 11th iteration of «`.*?,`» has consumed „`11,12,`". You can already see the root of the problem: the part of the regex (the dot) matching the contents of the field also matches the delimiter (the comma). Because of the double repetition (star inside {11}), this leads to a catastrophic amount of backtracking.

The regex engine now checks whether the 13th field starts with a P. It does not. Since there is no comma after the 13th field, the regex engine can no longer match the 11th iteration of «`.*?,`». But it does not give up there. It backtracks to the 10th iteration, expanding the match of the 10th iteration to „`10,11,`". Since there is still no P, the 10th iteration is expanded to „`10,11,12,`". Reaching the end of the string again, the same story starts with the 9th iteration, subsequently expanding it to „`9,10,`", „`9,10,11,`", „`9,10,11,12,`". But between each expansion, there are more possiblities to be tried. When the 9th iteration consumes „`9,10,`", the 10th could match just „`11,`" as well as „`11,12,`". Continuously failing, the engine backtracks to the 8th iteration, again trying all possible combinations for the 9th, 10th, and 11th iterations.

You get the idea: the possible number of combinations that the regex engine will try for each line where the 12th field does not start with a P is huge. This causes software like EditPad Pro to stop responding longer than your patience lasts. Other applications may even crash as the regex engine runs out of memory trying to remember all backtracking positions.

Preventing Catastrophic Backtracking

The solution is simple. When nesting repetition operators, make absolutely sure that there is only one way to match the same match. If repeating the inner loop 4 times and the outer loop 7 times results in the same overall match as repeating the inner loop 6 times and the outer loop 2 times, you can be sure that the regex engine will try all those combinations.

In our example, the solution is to be more exact about what we want to match. We want to match 11 comma-delimited fields. The fields must not contain comma's. So the regex becomes: «`^([^,\r\n]*,){11}P`» . If the P cannot be found, the engine will still backtrack. But it will backtrack only 11 times, and each time the «`[^,\r\n]`» is not able to expand beyond the comma, forcing the regex engine to the previous one of the 11 iterations immediately, without trying further options.

See the Difference with RegexBuddy

If you try this example with RegexBuddy's debugger, you will see that the original regex «`^(.*?,){11}P`» needs 48,066 steps to conclude there regex cannot match "`1,2,3,4,5,6,7,8,9,10,11,12,13`". If the string is "`1,2,3,4,5,6,7,8,9,10,11,12,13,14`", just 3 characters more, the number of steps jumps to 96,798. It's not too hard to imagine that at this kind of exponential rate, attempting this regex on a large file with long lines could easily take forever. RegexBuddy's debugger will abort the attempt after 100,000 steps, to prevent it from running out of memory.

Our improved regex «`^([^,\r\n]*,){11}P`», however, needs just twenty-six steps to fail, whether the subject string has 13 numbers or 14. While the complexity of the original regex was exponential, the complexity of the improved regex is constant.

Atomic Grouping and Possessive Quantifiers

Recent regex flavors have introduced two additional solutions to this problem: atomic grouping and possessive quantifiers. Their purpose is to prevent backtracking, allowing the regex engine to fail faster.

In the above example, we could easily reduce the amount of backtracking to a very low level by better specifying what we wanted. But that is not always possible in such a straightforward manner. In that case, you should use atomic grouping to prevent the regex engine from backtracking.

Using atomic grouping, the above regex becomes «`^(?>(.*?,){11})P`». Everything between `(?>)` is treated as one single token by the regex engine, once the regex engine leaves the group. Because the entire group is one token, no backtracking can take place once the regex engine has found a match for the group. If backtracking is required, the engine has to backtrack to the regex token before the group (the caret in our example). If there is no token before the group, the regex must retry the entire regex at the next position in the string.

Possessive quantifiers are a limited form of atomic grouping with a cleaner notation. To make a quantifier possessive, place a plus after it. «`x++`» is the same as «`(?>x+)`». Similarly, you can use «`x*+`», «`x?+`» and «`x{m,n}+`». Note that you cannot make a lazy quantifier possessive. It would match the minimum number of matches and never expand the match because backtracking is not allowed.

Tool and Language Support for Atomic Grouping and Possessive Quantifiers

Atomic grouping is a recent addition to the regex scene, and only supported by a small number of recent regex flavors. Perl supports it starting with version 5.6. Java supports it starting with JDK version 1.4.2, though the JDK documentation uses the term “independent group” rather than “atomic group”. All versions of .NET support atomic grouping, as do recent versions of PCRE, PHP's pgreg functions and Ruby. Python and VBScript do not support atomic grouping.

At this time, possessive quantifiers are only supported by the Java JDK 1.4.0 and later, and PCRE version 4 and later.

The latest versions of EditPad Pro and PowerGREP support both atomic grouping and possessive quantifiers, as do all versions of RegexBuddy.

Atomic Grouping Inside The Regex Engine

Let's see how «`^(?>(.*?,){11})P`» is applied to “`1,2,3,4,5,6,7,8,9,10,11,12,13`”. The caret matches at the start of the string and the engine enters the atomic group. The star is lazy, so the dot is initially skipped. But the comma does not match “`1`”, so the engine backtracks to the dot. That's right: backtracking is allowed here. The star is not possessive, and is not immediately enclosed by an atomic group. That is, the regex engine did not cross the closing round bracket of the atomic group. The dot matches „`1`”, and the comma matches too. «`{11}`» causes further repetition until the atomic group has matched „`1,2,3,4,5,6,7,8,9,10,11,`”.

Now, the engine leaves the atomic group. Because the group is atomic, all backtracking information is discarded and the group is now considered a single token. The engine now tries to match «`P`» to the “`1`” in the 12th field. This fails.

So far, everything happened just like in the original, troublesome regular expression. Now comes the difference. «`P`» failed to match, so the engine backtracks. The previous token is an atomic group, so the group's entire match is discarded and the engine backtracks further to the caret. The engine now tries to match the caret at the next position in the string, which fails. The engine walks through the string until the end, and declares failure. Failure is declared after 30 attempts to match the caret, and just one attempt to match the atomic group, rather than after 30 attempts to match the caret and a huge number of attempts to try all combinations of both quantifiers in the regex.

That is what atomic grouping and possessive quantifiers are for: efficiency by disallowing backtracking. The most efficient regex for our problem at hand would be «`^(?>((?>[^,\r\n]*),){11})P`» , since possessive, greedy repetition of the star is faster than a backtracking lazy dot. If possessive quantifiers are available, you can reduce clutter by writing «`^(?>([^,\r\n]*+,){11})P`» .

When To Use Atomic Grouping or Possessive Quantifiers

Atomic grouping and possessive quantifiers speed up failure by eliminating backtracking. They do not speed up success, only failure. When nesting quantifiers like in the above example, you really should use atomic grouping and/or possessive quantifiers whenever possible. While «`x[^x]*+x`» and «`x(?>[^x]*)x`» fail faster than «`x[^x]*x`», the increase in speed is minimal. If the final x in the regex cannot be matched, the regex

engine backtracks once for each character matched by the star. With simple repetition, the amount of time wasted with pointless backtracking increases in a linear fashion to the length of the string. With combined repetition, the amount of time wasted increases exponentially and will very quickly exhaust the capabilities of your computer. Still, if you are smart about combined repetition, you often can avoid the problem without atomic grouping as in the example above.

If you are simply doing a search in a text editor, using simple repetition like in the above paragraph, you will not earn back the extra time to type in the characters for the atomic grouping. If the regex will be used in a tight loop in an application, or process huge amounts of data, then atomic grouping may make a difference.

Note that atomic grouping and possessive quantifiers can alter the outcome of the regular expression match. «`\d+6`» will match „`123456`” in “`123456789`”. «`\d++6`» will not match at all. «`\d+`» will match the entire string. With the former regex, the engine backtracks until the 6 can be matched. In the latter case, no backtracking is allowed, and the match fails. Again, the cause of this is that the token «`\d`» that is repeated can also match the delimiter «`6`». Sometimes this is desirable, often it is not.

This shows again that understanding how the regex engine works on the inside will enable you to avoid many pitfalls and craft efficient regular expressions that match exactly what you want.

However, the most important situation in which to use atomic grouping is when you cannot use a negated character class to prevent a repeated regex token (typically used to match the content of something) from matching the non-repeated regex token that follows it (typically used to match the content's delimiter). The following example illustrates this.

Quickly Matching a Complete HTML File

Suppose you want to use a regular expression to match a complete HTML file, and extract the basic parts from the file. If you know the structure of HTML files, writing the regex «`<html>.*?<head>.*?<title>.*?</title>.*?</head>.*?<body[^>]*>.*?</body>.*?</html>`» is very straight-forward. With the “dot matches newlines” or “single line” matching mode turned on, it will work just fine on valid HTML files.

Unfortunately, this regular expression won't work nearly as well on an HTML file that misses some of the tags. The worst case is a missing </html> tag at the end of the file. When «`<html>`» fails to match, the regex engine backtracks, giving up the match for «`</body>.*?`». It will then further expand the lazy dot before «`</body>`», looking for a second closing “`</body>`” tag in the HTML file. When that fails, the engine gives up «`<body[^>]*>.*?`», and starts looking for a second opening “`<body[^>]*>`” tag all the way to the end of the file. Since that also fails, the engine proceeds looking all the way to the end of the file for a second closing head tag, a second closing title tag, etc.

If you run this regex in RegexBuddy's debugger, the output will look like a sawtooth. The regex matches the whole file, backs up a little, matches the whole file again, backs up some more, backs up yet some more, matches everything again, etc. until each of the 7 «`.*?`» tokens has reached the end of the file. The result is that this regular has a worst case complexity of N^7. If you double the length of the HTML file with the missing <html> tag by appending text at the end, the regular expression will take 128 times (2^7) as long to figure out the HTML file isn't valid.

In this situation, we know that each of the literal text blocks in our regular expression (the HTML tags, which function as delimiters) will occur only once in a valid HTML file. That makes it very easy to package each of the lazy dots (the delimited content) in an atomic group.

«`<html>(?>.*?<head>)(?>.*?<title>)(?>.*?</title>)(?>.*?</head>)(?>.*?<body[^>]*>)(?>.*?</body>).*?</html>`» will match a valid HTML file in the same number of steps as the original regex. The gain is that it will fail on an invalid HTML file almost as fast as it matches a valid one. When «`<html>`» fails to match, the regex engine backtracks, giving up the match for the last lazy dot. But then, there's nothing further to backtrack to. Since all of the lazy dots are in an atomic group, the regex engines has discarded their backtracking positions. The groups function as a "do not expand further" roadblock. The regex engine is forced to announce failure immediately.

I'm sure you've noticed that each atomic group also contains an HTML tag after the lazy dot. This is critical. We do allow the lazy dot to backtrack until its matching HTML tag was found. E.g. when «`.*?</body>`» is processing "`Last paragraph</p></body>`", the «`</`» regex tokens will match „`</`" in "`</p>`". However, «`b`» will fail "`p`". At that point, the regex engine will backtrack and expand the lazy dot to include „`</p>`". Since the regex engine hasn't left the atomic group yet, it is free to backtrack inside the group. Once «`</body>`» has matched, and the regex engine leaves the atomic group, it discards the lazy dot's backtracking positions. Then it can no longer be expanded.

Essentially, what we've done is bind a repeated regex token (the lazy dot to match HTML content) to the non-repeated regex token that follows it (the literal HTML tag). Since anything, including HTML tags, can appear between the HTML tags in our regular expression, we cannot use a negated character class instead of the lazy dot to prevent the delimiting HTML tags from being matched as HTML content. But we can and did achieve the same result by combining each lazy dot and the HTML tag following it into an atomic group. As soon as the HTML tag is matched, the lazy dot's match is locked down. This ensures that the lazy dot will never match the HTML tag that should be matched by the literal HTML tag in the regular expression.

16. Lookahead and Lookbehind Zero-Width Assertions

Perl 5 introduced two very powerful constructs: “lookahead” and “lookbehind”. Collectively, these are called “lookaround”. They are also called “zero-width assertions”. They are zero-width just like the start and end of line, and start and end of word anchors that I already explained. The difference is that lookarounds will actually match characters, but then give up the match and only return the result: match or no match. That is why they are called “assertions”. They do not consume characters in the string, but only assert whether a match is possible or not.

Lookarounds allow you to create regular expressions that are impossible to create without them, or that would get very longwinded without them. All regex flavors discussed in this book support lookaround. The exception is JavaScript, which supports lookahead but not lookbehind.

Positive and Negative Lookahead

Negative lookahead is indispensable if you want to match something not followed by something else. When explaining character classes, I already explained why you cannot use a negated character class to match a “q” not followed by a “u”. Negative lookahead provides the solution: «`q(?!u)`». The negative lookahead construct is the pair of round brackets, with the opening bracket followed by a question mark and an exclamation point. Inside the lookahead, we have the trivial regex «u».

Positive lookahead works just the same. «`q(?=u)`» matches a q that is followed by a u, without making the u part of the match. The positive lookahead construct is a pair of round brackets, with the opening bracket followed by a question mark and an equals sign.

You can use any regular expression inside the lookahead. (Note that this is not the case with lookbehind. I will explain why below.) Any valid regular expression can be used inside the lookahead. If it contains capturing parentheses, the backreferences will be saved. Note that the lookahead itself does not create a backreference. So it is not included in the count towards numbering the backreferences. If you want to store the match of the regex inside a backreference, you have to put capturing parentheses around the regex inside the lookahead, like this: «`(?=(regex))`». The other way around will not work, because the lookahead will already have discarded the regex match by the time the backreference is to be saved.

Regex Engine Internals

First, let's see how the engine applies «`q(?!u)`» to the string “`Iraq`”. The first token in the regex is the literal «q». As we already know, this will cause the engine to traverse the string until the „q” in the string is matched. The position in the string is now the void behind the string. The next token is the lookahead. The engine takes note that it is inside a lookahead construct now, and begins matching the regex inside the lookahead. So the next token is «u». This does not match the void behind the string. The engine notes that the regex inside the lookahead failed. Because the lookahead is negative, this means that the lookahead has successfully matched at the current position. At this point, the entire regex has matched, and „q” is returned as the match.

Let's try applying the same regex to “`quit`”. «q» matches „q”. The next token is the «u» inside the lookahead. The next character is the “u”. These match. The engine advances to the next character: “`i`”. However, it is done with the regex inside the lookahead. The engine notes success, and discards the regex match. This causes the engine to step back in the string to “u”.

Because the lookahead is negative, the successful match inside it causes the lookahead to fail. Since there are no other permutations of this regex, the engine has to start again at the beginning. Since «q» cannot match anywhere else, the engine reports failure.

Let's take one more look inside, to make sure you understand the implications of the lookahead. Let's apply «q(?=u)i» to "quit". I have made the lookahead positive, and put a token after it. Again, «q» matches „q" and «u» matches „u". Again, the match from the lookahead must be discarded, so the engine steps back from "i" in the string to "u". The lookahead was successful, so the engine continues with «i». But «i» cannot match "u". So this match attempt fails. All remaining attempts will fail as well, because there are no more q's in the string.

Positive and Negative Lookbehind

Lookbehind has the same effect, but works backwards. It tells the regex engine to temporarily step backwards in the string, to check if the text inside the lookbehind can be matched there. «(?<!a)b» matches a "b" that is not preceded by an "a", using negative lookbehind. It will not match "cab", but will match the „b" (and only the „b") in "bed" or "debt". «(?<=a)b» (positive lookbehind) matches the „b" (and only the „b") in „cab", but does not match "bed" or "debt".

The construct for positive lookbehind is «(?<=text)»: a pair of round brackets, with the opening bracket followed by a question mark, "less than" symbol and an equals sign. Negative lookbehind is written as «(?<!text)», using an exclamation point instead of an equals sign.

More Regex Engine Internals

Let's apply «(?<=a)b» to "thingamabob". The engine starts with the lookbehind and the first character in the string. In this case, the lookbehind tells the engine to step back one character, and see if an "a" can be matched there. The engine cannot step back one character because there are no characters before the "t". So the lookbehind fails, and the engine starts again at the next character, the "h". (Note that a negative lookbehind would have succeeded here.) Again, the engine temporarily steps back one character to check if an "a" can be found there. It finds a "t", so the positive lookbehind fails again.

The lookbehind continues to fail until the regex reaches the "m" in the string. The engine again steps back one character, and notices that the „a" can be matched there. The positive lookbehind matches. Because it is zero-width, the current position in the string remains at the "m". The next token is «b», which cannot match here. The next character is the second "a" in the string. The engine steps back, and finds out that the "m" does not match «a».

The next character is the first "b" in the string. The engine steps back and finds out that „a" satisfies the lookbehind. «b» matches „b", and the entire regex has been matched successfully. It matches one character: the first „b" in the string.

Important Notes About Lookbehind

The good news is that you can use lookbehind anywhere in the regex, not only at the start. If you want to find a word not ending with an "s", you could use «\b\w+(?<!s)\b». This is definitely not the same as

«\b\w+[^s]\b». When applied to "John's", the former will match „John" and the latter „John'" (including the apostrophe). I will leave it up to you to figure out why. (Hint: «\b» matches between the apostrophe and the "s"). The latter will also not match single-letter words like "a" or "I". The correct regex without using lookbehind is «\b\w*[^s\W]\b» (star instead of plus, and \W in the character class). Personally, I find the lookbehind easier to understand. The last regex, which works correctly, has a double negation (the \W in the negated character class). Double negations tend to be confusing to humans. Not to regex engines, though.

The bad news is that most regex flavors do not allow you to use just any regex inside a lookbehind, because they cannot apply a regular expression backwards. Therefore, the regular expression engine needs to be able to figure out how many steps to step back before checking the lookbehind.

Therefore, many regex flavors, including those used by Perl 5 and Python, only allow fixed-length strings. You can use any regex of which the length of the match can be predetermined. This means you can use literal text and character classes. You cannot use repetition or optional items. You can use alternation, but only if all options in the alternation have the same length.

Some regex flavors support the above, plus alternation with strings of different lengths. But each string in the alternation must still be of fixed length, so only literals and character classes can be used. This includes PCRE, PHP and EditPad Pro.

More advanced flavors support the above, plus finite repetition. This means you can still not use the star or plus, but you can use the question mark and the curly braces with the max parameter specified. These regex flavors recognize the fact that finite repetition can be rewritten as an alternation of strings with different, but fixed lengths. The only regex flavor that I know of that currently supports this is Sun's regex package in the JDK 1.4.

The only regex flavors that allow you to use a full regular expression inside lookbehind are those used by RegexBuddy (2.0.3 and later), PowerGREP (3.0.0 and later), and the .NET framework (all versions).

Finally, JavaScript, Ruby and VBScript do not support lookbehind at all.

Lookaround Is Atomic

The fact that lookaround is zero-width automatically makes it atomic. As soon as the lookaround condition is satisfied, the regex engine forgets about everything inside the lookaround. It will not backtrack inside the lookaround to try different permutations.

The only situation in which this makes any difference is when you use capturing groups inside the lookaround. Since the regex engine does not backtrack into the lookaround, it will not try different permutations of the capturing groups.

For this reason, the regex «(?=(\d+))\w+\1» will never match "123x12". First the lookaround captures „123" into «\1». «\w+» then matches the whole string and backtracks until it matches only „1". Finally, «\w+» fails since «\1» cannot be matched at any position. Now, the regex engine has nothing to backtrack to, and the overall regex fails. The backtracking steps created by «\d+» have been discarded. It never gets to the point where the lookahead captures only "12".

Obviously, the regex engine does try further positions in the string. If we change the subject string, the regex «`(?=(\d+))\w+\1`» will match „`56x56`” in “`456x56`”.

If you don't use capturing groups inside lookaround, then all this doesn't matter. Either the lookaround condition can be satisfied or it cannot be. In how many ways it can be satisfied is irrelevant.

17. Testing The Same Part of The String for More Than One Requirement

Lookaround, which I introduced in detail in the previous topic, is a very powerful concept. Unfortunately, it is often underused by people new to regular expressions, because lookaround is a bit confusing. The confusing part is that the lookaround is zero-width. So if you have a regex in which a lookahead is followed by another piece of regex, or a lookbehind is preceded by another piece of regex, then the regex will traverse part of the string twice.

To make this clear, I would like to give you another, a bit more practical example. Let's say we want to find a word that is six letters long and contains the three subsequent letters "`cat`". Actually, we can match this without lookaround. We just specify all the options and hump them together using alternation: «`cat\w{3}|\wcat\w{2}|\w{2}cat\w|\w{3}cat`». Easy enough. But this method gets unwieldy if you want to find any word between 6 and 12 letters long containing either "cat", "dog" or "mouse".

Lookaround to The Rescue

In this example, we basically have two requirements for a successful match. First, we want a word that is 6 letters long. Second, the word we found must contain the word "cat".

Matching a 6-letter word is easy with «`\b\w{6}\b`». Matching a word containing "cat" is equally easy: «`\b\w*cat\w*\b`».

Combining the two, we get: «`(?=\b\w{6}\b)\b\w*cat\w*\b`» . Easy! Here's how this works. At each character position in the string where the regex is attempted, the engine will first attempt the regex inside the positive lookahead. This sub-regex, and therefore the lookahead, matches only when the current character position in the string is at the start of a 6-letter word in the string. If not, the lookahead will fail, and the engine will continue trying the regex from the start at the next character position in the string.

The lookahead is zero-width. So when the regex inside the lookahead has found the 6-letter word, the current position in the string is still at the beginning of the 6-letter word. At this position will the regex engine attempt the remainder of the regex. Because we already know that a 6-letter word can be matched at the current position, we know that «`\b`» matches and that the first «`\w*`» will match 6 times. The engine will then backtrack, reducing the number of characters matched by «`\w*`», until «`cat`» can be matched. If «`cat`» cannot be matched, the engine has no other choice but to restart at the beginning of the regex, at the next character position in the string. This is at the second letter in the 6-letter word we just found, where the lookahead will fail, causing the engine to advance character by character until the next 6-letter word.

If «`cat`» can be successfully matched, the second «`\w*`» will consume the remaining letters, if any, in the 6-letter word. After that, the last «`\b`» in the regex is guaranteed to match where the second «`\b`» inside the lookahead matched. Our double-requirement-regex has matched successfully.

Optimizing Our Solution

While the above regex works just fine, it is not the most optimal solution. This is not a problem if you are just doing a search in a text editor. But optimizing things is a good idea if this regex will be used repeatedly and/or on large chunks of data in an application you are developing.

You can discover these optimizations by yourself if you carefully examine the regex and follow how the regex engine applies it, as I did above. I said the third and last «`\b`» are guaranteed to match. Since it is zero-width, and therefore does not change the result returned by the regex engine, we can remove them, leaving: «`(?=\b\w{6}\b)\w*cat\w*`». Though the last «`\w*`» is also guaranteed to match, we cannot remove it because it adds characters to the regex match. Remember that the lookahead discards its match, so it does not contribute to the match returned by the regex engine. If we omitted the «`\w*`», the resulting match would be the start of a 6-letter word containing "cat", up to and including "cat", instead of the entire word.

But we can optimize the first «`\w*`». As it stands, it will match 6 letters and then backtrack. But we know that in a successful match, there can never be more than 3 letters before "cat". So we can optimize this to «`\w{0,3}`». Note that making the asterisk lazy would not have optimized this sufficiently. The lazy asterisk would find a successful match sooner, but if a 6-letter word does not contain "cat", it would still cause the regex engine to try matching "cat" at the last two letters, at the last single letter, and even at one character beyond the 6-letter word.

So we have «`(?=\b\w{6}\b)\w{0,3}cat\w*`». One last, minor, optimization involves the first «`\b`». Since it is zero-width itself, there's no need to put it inside the lookahead. So the final regex is: «`\b(?=\w{6}\b)\w{0,3}cat\w*`» .

A More Complex Problem

So, what would you use to find any word between 6 and 12 letters long containing either "cat", "dog" or "mouse"? Again we have two requirements, which we can easily combine using a lookahead: «`\b(?=\w{6,12}\b)\w{0,9}(cat|dog|mouse)\w*`» . Very easy, once you get the hang of it. This regex will also put "cat", "dog" or "mouse" into the first backreference.

18. Continuing at The End of The Previous Match

The anchor «\G» matches at the position where the previous match ended. During the first match attempt, «\G» matches at the start of the string in the way «\A» does.

Applying «\G\w» to the string "test string" matches „t". Applying it again matches „e". The 3rd attempt yields „s" and the 4th attempt matches the second „t" in the string. The fifth attempt fails. During the fifth attempt, the only place in the string where «\G» matches is after the second t. But that position is not followed by a word character, so the match fails.

End of The Previous Match vs Start of The Match Attempt

With some regex flavors or tools, «\G» matches at the start of the match attempt, rather than at the end of the previous match result. This is the case with EditPad Pro, where «\G» matches at the position of the text cursor, rather than the end of the previous match. When a match is found, EditPad Pro will select the match, and move the text cursor to the end of the match. The result is that «\G» matches at the end of the previous match result only when you do not move the text cursor between two searches. All in all, this makes a lot of sense in the context of a text editor.

\G Magic with Perl

In Perl, the position where the last match ended is a "magical" value that is remembered separately for each string variable. The position is not associated with any regular expression. This means that you can use «\G» to make a regex continue in a subject string where another regex left off.

If a match attempt fails, the stored position for «\G» is reset to the start of the string. To avoid this, specify the continuation modifier /c.

All this is very useful to make several regular expressions work together. E.g. you could parse an HTML file in the following fashion:

```
while ($string =~ m/</g) {
  if ($string =~ m/\GB>/c) {
    # Bold
  } elsif ($string =~ m/\GI>/c) {
    # Italics
  } else {
    # ...etc...
  }
}
```

The regex in the while loop searches for the tag's opening bracket, and the regexes inside the loop check which tag we found. This way you can parse the tags in the file in the order they appear in the file, without having to write a single big regex that matches all tags you are interested in.

\G in Other Programming Langauges

This flexibility is not available with most other programming languages. E.g. in Java, the position for «\G» is remembered by the Matcher object. The Matcher is strictly associated with a single regular expression and a single subject string. What you can do though is to add a line of code to make the match attempt of the second Matcher start where the match of the first Matcher ended. «\G» will then match at this position.

19. If-Then-Else Conditionals in Regular Expressions

A special construct «`(?ifthen|else)`» allows you to create conditional regular expressions. If the *if* part evaluates to true, then the regex engine will attempt to match the *then* part. Otherwise, the *else* part is attempted instead. The syntax consists of a pair of round brackets. The opening bracket must be followed by a question mark, immediately followed by the *if* part, immediately followed by the *then* part. This part can be followed by a vertical bar and the *else* part. You may omit the *else* part, and the vertical bar with it.

For the *if* part, you can use the lookahead and lookbehind constructs. Using positive lookahead, the syntax becomes «`(?(?=regex)then|else)`». Because the lookahead has its own parentheses, the *if* and *then* parts are clearly separated.

Remember that the lookaround constructs do not consume any characters. If you use a lookahead as the *if* part, then the regex engine will attempt to match the *then* or *else part* (depending on the outcome of the lookahead) at the same position where the *if* was attempted.

Alternatively, you can check in the *if* part whether a capturing group has taken part in the match thus far. Place the number of the capturing group inside round brackets, and use that as the if part. Note that although the syntax for a conditional check on a backreference is the same as a number inside a capturing groups, no capturing groups is created. The number and the brackets are part of the if-then-else syntax started with «`(?`».

The regex «`(a)?b(?(1)c|d)`» matches „`bd`" and „`abc`". If the a can be matched, the first capturing group takes part in the match. The conditional then uses the *then* part «`c`». Otherwise, the *else* part «`d`» is used.

For the *then* and *else*, you can use any regular expression. If you want to use alternation, you will have to group the *then* or *else* together using parentheses, like in «`(?(?=condition)(then1|then2|then3)|(else1|else2|else3))`». Otherwise, there is no need to use parentheses around the *then* and *else* parts.

Regex Flavors

Conditionals are supported by PCRE, the PHP preg functions and the .NET framework. The latest versions of EditPad Pro, PowerGREP and RegexBuddy also support them.

The .NET framework allows you to use the name of a named capturing group for the *if* test, e.g.: «`(?<test>a)?b(?(test)c|d)`». None of the other flavors, though they all support named capture, allow the name of a group as a conditional test. You have to use the number.

Example: Extract Email Headers

The regex «`^((From|To)|Subject): ((?(2)\w+@\w+\.[a-z]+|.+))`» extracts the From, To, and Subject headers from an email message. The name of the header is captured into the first backreference. If the header is the From or To header, it is captured into the second backreference as well.

The second part of the pattern is the if-then-else conditional «`(?(2)\w+@\w+\.[a-z]+|.+))`». The if part checks if the second capturing group took part in the match thus far. It will have if the header is the From or To header. In that case, we the *then* part of the conditional «`\w+@\w+\.[a-z]+`» tries to match an email

address. To keep the example simple, we use an overly simple regex to match the email address, and we don't try to match the display name that is usually also part of the From or To header.

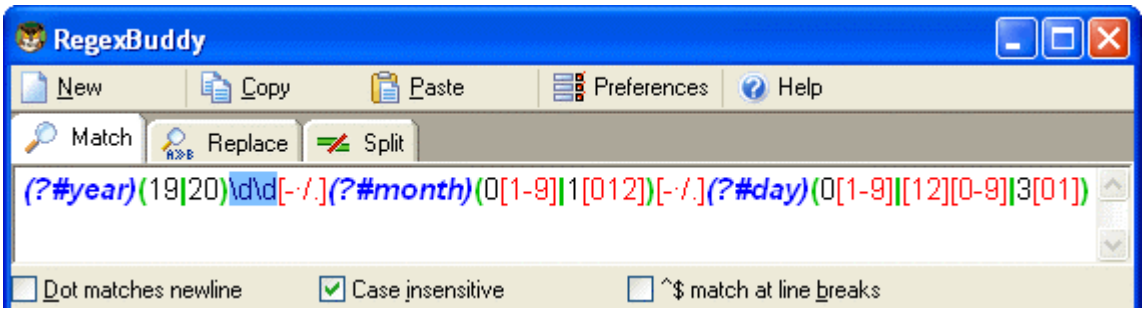
If the second capturing group did not participate in the match this far, the *else* part «.+» is attempted instead.

Finally, we place an extra pair of round brackets around the conditional. This captures the contents of the email header matched by the conditional into the third backreference. The conditional itself does not capture anything.

20. Adding Comments to Regular Expressions

If you have worked through the entire tutorial, I guess you will agree that regular expressions can quickly become rather cryptic. Therefore, many modern regex flavors allow you to insert comments into regexes. The syntax is «`(?#comment)`» where "comment" can be whatever you want, as long as it does not contain a closing round bracket. The regex engine ignores everything after the «`(?#`» until the first closing round bracket.

E.g. I could clarify the regex to match a valid date by writing it as «`(?#year)(19|20)\d\d[-/.](?#month)(0[1-9]|1[012])[- /.](?#day)(0[1-9]|[12][0-9]|3[01])`» . Now it is instantly obvious that this regex matches a date in yyyy-mm-dd format. Some software, such as RegexBuddy, EditPad Pro and PowerGREP can apply syntax coloring to regular expressions while you write them. That makes the comments really stand out, enabling the right comment in the right spot to make a complex regular expression much easier to understand.



Part 3

Regular Expression Examples

1. Sample Regular Expressions

Below, you will find many example patterns that you can use for and adapt to your own purposes. Key techniques used in crafting each regex are explained, with links to the corresponding pages in the tutorial where these concepts and techniques are explained in great detail.

If you are new to regular expressions, you can take a look at these examples to see what is possible. Regular expressions are very powerful. They do take some time to learn. But you will earn back that time quickly when using regular expressions to automate searching or editing tasks in EditPad Pro or PowerGREP, or when writing scripts or applications in a variety of languages.

RegexBuddy offers the fastest way to get up to speed with regular expressions. RegexBuddy will analyze any regular expression and present it to you in a clearly to understand, detailed outline. The outline links to RegexBuddy's regex tutorial (the same one you find on this web site), where you can always get in-depth information with a single click.

Oh, and you definitely do not need to be a programmer to take advantage of regular expressions!

Grabbing HTML Tags

«`<TAG[^>]*>(.*?)</TAG>`» matches the opening and closing pair of a specific HTML tag. Anything between the tags is captured into the first backreference. The question mark in the regex makes the star lazy, to make sure it stops before the first closing tag rather than before the last, like a greedy star would do. This regex will not properly match tags nested inside themselves, like in "`<TAG>one<TAG>two</TAG>one</TAG>`".

«`<([A-Z][A-Z0-9]*)[^>]*>(.*?)</\1>`» will match the opening and closing pair of any HTML tag. Be sure to turn off case sensitivity. The key in this solution is the use of the backreference «`\1`» in the regex. Anything between the tags is captured into the second backreference. This solution will also not match tags nested in themselves.

Trimming Whitespace

You can easily trim unnecessary whitespace from the start and the end of a string or the lines in a text file by doing a regex search-and-replace. Search for «`^[ \t]+`» and replace with nothing to delete leading whitespace (spaces and tabs). Search for «`[ \t]+$`» to trim trailing whitespace. Do both by combining the regular expressions into «`^[ \t]+|[ \t]+$`» . Instead of `[ \t]` which matches a space or a tab, you can expand the character class into «`[ \t\r\n]`» if you also want to strip line breaks. Or you can use the shorthand «`\s`» instead.

IP Addresses

Matching an IP address is another good example of a trade-off between regex complexity and exactness. «`\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b`» will match any IP address just fine, but will also match „`999.999.999.999`" as if it were a valid IP address. Whether this is a problem depends on the files or data you intend to apply the regex to. To restrict all 4 numbers in the IP address to 0..255, you can use this complex beast: «`\b(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.`»«`(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.`»«`(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.`»«`(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b`» (everything on a single line). The long regex stores each of the 4 numbers of the IP address into a capturing group. You can use these groups to further process the IP number.

If you don't need access to the individual numbers, you can shorten the regex with a quantifier to: «`\b(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}`»«`(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b`» . Similarly, you can shorten the quick regex to «`\b(?:\d{1,3}\.){3}\d{1,3}\b`»

More Detailed Examples

Matching a Floating Point Number. Also illustrates the common mistake of making everything in a regular expression optional.

Matching an Email Address. There's a lot of controversy about what is a proper regex to match email addresses. It's a perfect example showing that you need to know exactly what you're trying to match (and what not), and that there's always a trade-off between regex complexity and accuracy.

Matching Valid Dates. A regular expression that matches 31-12-1999 but not 31-13-1999.

Matching Complete Lines. Shows how to match complete lines in a text file rather than just the part of the line that satisfies a certain requirement. Also shows how to match lines in which a particular regex does *not* match.

Removing Duplicate Lines or Items. Illustrates simple yet clever use of capturing parentheses or backreferences.

Regex Examples for Processing Source Code. How to match common programming language syntax such as comments, strings, numbers, etc.

Two Words Near Each Other. Shows how to use a regular expression to emulate the "near" operator that some tools have.

2. Matching Floating Point Numbers with a Regular Expression

In this example, I will show you how you can avoid a common mistake often made by people inexperienced with regular expressions. As an example, we will try to build a regular expression that can match any floating point number. Our regex should also match integers, and floating point numbers where the integer part is not given (i.e. zero). We will not try to match numbers with an exponent, such as 1.5e8 (150 million in scientific notation).

At first thought, the following regex seems to do the trick: «`[-+]?[0-9]*\.?[0-9]*`». This defines a floating point number as an optional sign, followed by an optional series of digits (integer part), followed by an optional dot, followed by another optional series of digits (fraction part).

Spelling out the regex in words makes it obvious: everything in this regular expression is optional. This regular expression will consider a sign by itself or a dot by itself as a valid floating point number. In fact, it will even consider an empty string as a valid floating point number. This regular expression can cause serious trouble if it is used in a scripting language like Perl or PHP to verify user input.

Not escaping the dot is also a common mistake. A dot that is not escaped will match any character, including a dot. If we had not escaped the dot, "`4.4`" would be considered a floating point number, and "`4X4`" too.

When creating a regular expression, it is more important to consider what it should *not* match, than what it should. The above regex will indeed match a proper floating point number, because the regex engine is greedy. But it will also match many things we do not want, which we have to exclude.

Here is a better attempt: «`[-+]?([0-9]*\.[0-9]+|[0-9]+)`». This regular expression will match an optional sign, that is either followed by zero or more digits followed by a dot and one or more digits (a floating point number with optional integer part), or followed by one or more digits (an integer).

This is a far better definition. Any match will include at least one digit, because there is no way around the «`[0-9]+`» part. We have successfully excluded the matches we do not want: those without digits.

We can optimize this regular expression as: «`[-+]?[0-9]*\.?[0-9]+`» .

If you also want to match numbers with exponents, you can use: «`[-+]?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?`» . Notice how I made the entire exponent part optional by grouping it together, rather than making each element in the exponent optional.

3. How to Find or Validate an Email Address

The regular expression I receive the most feedback, not to mention “bug” reports on, is the one you'll find right in the tutorial's introduction: «`\b[A-Z0-9._%-]+@[A-Z0-9-]+\.[A-Z]{2,4}\b`» . This regular expression, I claim, matches any email address. Most of the feedback I get refutes that claim by showing one email address that this regex doesn't match. Usually, the “bug” report also includes a suggestion to make the regex “perfect”.

As I explain below, my claim only holds true when one accepts my definition of what a valid email address really is, and what it's not. If you want to use a different definition, you'll have to adapt the regex. I'll also show you that a regex to match truly *any* possible email address is not only hideously complex, it's also totally useless. Matching a valid email address is a perfect example showing that (1) before writing a regex, you have to know exactly what you're trying to match, and what not; and (2) there's often a trade-off between what's exact, and what's practical.

The virtue of my regular expression above is that it matches 99% of the email addresses in use today. All the email address it matches can be handled by 99% of all email software out there. If you're looking for a quick solution, you only need to read the next paragraph. If you want to know all the trade-offs and get plenty of alternatives to choose from, read on.

If you want to use the regular expression above, there's two things you need to understand. First, long regexes make it difficult to nicely format paragraphs. So I didn't include «`a-z`» in any of the three character classes. This regex is intended to be used with your regex engine's “case insensitive” option turned on. (You'd be surprised how many “bug” reports I get about that.) Second, the above regex is delimited with word boundaries, which makes it suitable for extracting email addresses from files or larger blocks of text. If you want to check whether the user typed in a valid email address, replace the word boundaries with start-of-string and end-of-string anchors, like this: «`^[A-Z0-9._%-]+@[A-Z0-9-]+\.[A-Z]{2,4}$`».

Trade-Offs in Validating Email Addresses

Yes, there are a whole bunch of email addresses that my pet regex doesn't match. The most frequently quoted example are addresses on the `.museum` top level domain, which is longer than the 4 letters my regex allows for the top level domain. I accept this trade-off because the number of people using `.museum` email addresses is extremely low. I've never had a complaint that the order forms or newsletter subscription forms on the JGsoft web sites refused a `.museum` address (which they would, since they use the above regex to validate the email address).

To include `.museum`, you could use «`^[A-Z0-9._%-]+@[A-Z0-9-]+\.[A-Z]{2,6}$`». However, then there's another trade-off. This regex will match „`john@mail.office`”. It's far more likely that John forgot to type in the `.com` top level domain rather than having just created a new `.office` top level domain without ICANN's permission.

This shows another trade-off: do you want the regex to check if the top level domain exists? My regex doesn't. Any combination of two to four letters will do, which covers all existing and planned top level domains except .museum. But it will match addresses with invalid top-level domains like „`asdf@asdf.asdf`”. By not being overly strict about the top-level domain, I don't have to update the regex each time a new top-level domain is created, whether it's a country code or generic domain.

You could use «`^[A-Z0-9._%-]+@[A-Z0-9-]+\.([A-Z]{2}|com|org|net|biz|info|name|aero|biz|info|jobs|museum|name)$`» to allow any two-letter country code top level domain, and only specific generic top level domains. By the time you read this, the list might already be out of date. If you use this regular expression, I recommend you store it in a global constant in your application, so you only have to update it in one place. You could list all country codes in the same manner, even though there's almost 200 of them.

Another trade-off is that my regex only allows English letters, digits and a few special symbols. The main reason is that I don't trust all my email software to be able to handle much else. Even though `John.O'Hara@theharas.com` is a syntactically valid email address, there's a risk that some software will misinterpret the apostrophe as a delimiting quote. And of course, it's been many years already that domain names can include non-English characters. Most software and even domain name registrars, however, still stick to the 37 characters they're used to.

The conclusion is that to decide which regular expression to use, whether you're trying to match an email address or something else that's vaguely defined, you need to start with considering all the trade-offs. How bad is it to match something that's not valid? How bad is it not to match something that is valid? How complex can your regular expression be? How expensive would it be if you had to change the regular expression later? Different answers to these questions will require a different regular expression as the solution. My email regex does what I want, but it may not do what you want.

Regexes Don't Send Email

Don't go overboard in trying to eliminate invalid email addresses with your regular expression. If you have to accept `.museum` domains, allowing any 6-letter top level domain is often better than spelling out a list of all current domains. The reason is that you don't really know whether an address is valid until you try to send an email to it. And even that might not be enough. Even if the email arrives in a mailbox, that doesn't mean somebody still reads that mailbox.

The same principle applies in many situations. When trying to match a valid date, it's often easier to use a bit of arithmetic to check for leap years, rather than trying to do it in a regex. Use a regular expression to find potential matches or check if the input uses the proper syntax, and do the actual validation on the potential matches returned by the regular expression. Regular expressions are a powerful tool, but they're far from a panacea.

The Official Standard: RFC 822

Maybe you're wondering why there's no “official” full-proof and fool-proof regex to match email addresses. Well, there is an official definition, but it's hardly fool-proof, not to mention practical.

The official standard is known as RFC 822. The standard was written in 1982, long before the world wide web or the current domain system existed. A regular expression that follows RFC 822 to the letter would accept any currently valid email address. It would not skip odd-ball email addresses like my regex does. However, the RFC 822 regex would also accept many addresses that your desktop email software most likely cannot handle.

RFC 822 does not require email addresses to be fully qualified. That is, if I'm using a the computer with the domain name `downstairs.myhouse.th` and my wife's using `upstairs.myhouse.th`, then I can email her

using the email address `wife@upstairs`. Or at least, if I'm running my own email servers and my client knows how to deal with domains that aren't fully qualified. It's not going to work with Outlook Express.

However, I doubt that if I enter `wife@upstairs` on your web site's order form to buy her a gift, that your web site is going to be able to figure out how to reach her. That's why a regex implementing RFC 822 is useless. It's a tad bit long too:

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:
(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.
|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=
[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(
?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \
x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\
[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])
*(?:[^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[
\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*
))*|(?:[^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(
?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*
"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".
\[\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\
]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?
[ \t])*(?:[^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\
Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[
 \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \x00-\x1F
]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]
\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()
<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@
,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?
:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r\
n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?
:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[
^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()
<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\
r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \x00-\x
1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[
\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^
()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<
>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(
?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r
\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(
?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:
(?:[^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\
["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:
(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \
x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"
(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:
(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r\n)?
[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(
?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \x
00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[
([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\]
 \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))
|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(
?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:
\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*
\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\
[\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]
```

```
]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)
?[ \t])*(?:[^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|
\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?
[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \x00-\x1F]
+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\
r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^(
)<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>
@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\
n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \x00-\x
1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\
"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\
n)?[ \t])*(?:[^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])
+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n
)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \x00-\x1
F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\
]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)(?:,\s
*(?:(?:[^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(
?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*
"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[
\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]
))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*
@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r
\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\]
(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\
] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]])
)|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".
\[\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\
]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)
*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?
:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\
\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\
\".\[\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".
\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\
r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t
])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r
\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \x00-\
x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\
[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?
:[^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["
()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?
:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \x0
0-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?
:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?
:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \x00-\x1F]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \x00
-\x1F]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([
^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*))*
)?;\s*)
```

I broke this one up so it would fit on the page. But actually it's one long line of 6343 characters. It works fine though. If you want to try it in RegexBuddy, it's included in RegexBuddy's library.

4. Matching a Valid Date

«`(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])`» matches a date in yyyy-mm-dd format from between 1900-01-01 and 2099-12-31, with a choice of four separators. The year is matched by «`(19|20)\d\d`». I used alternation to allow the first two digits to be 19 or 20. The round brackets are mandatory. Had I omitted them, the regex engine would go looking for 19 or the remainder of the regular expression, which matches a date between 2000-01-01 and 2099-12-31. Round brackets are the only way to stop the vertical bar from splitting up the entire regular expression into two options.

The month is matched by «`0[1-9]|1[012]`», again enclosed by round brackets to keep the two options together. By using character classes, the first option matches a number between 01 and 09, and the second matches 10, 11 or 12.

The last part of the regex consists of three options. The first matches the numbers 01 through 09, the second 10 through 29, and the third matches 30 or 31.

Smart use of alternation allows us to exclude invalid dates such as 2000-00-00 that could not have been excluded without using alternation. To be really perfectionist, you would have to split up the month into various options to take into account the length of the month. The above regex still matches 2003-02-31, which is not a valid date. Making leading zeros optional could be another enhancement.

If you want to require the delimiters to be consistent, you could use a backreference. «`(19|20)\d\d([- /.])(0[1-9]|1[012])\2(0[1-9]|[12][0-9]|3[01])`» will match „`1999-01-01`” but not “`1999/01-01`”.

Again, how complex you want to make your regular expression depends on the data you are using it on, and how big a problem it is if an unwanted match slips through. If you are validating the user's input of a date in a script, it is probably easier to do certain checks outside of the regex. For example, excluding February 29th when the year is not a leap year is far easier to do in a scripting language. It is far easier to check if a year is divisible by 4 (and not divisible by 100 unless divisible by 400) using simple arithmetic than using regular expressions.

Here is how you could check a valid date in Perl. Note that I added anchors to make sure the entire variable is a date, and not a piece of text containing a date. I also added round brackets to capture the year into a backreference.

```
sub isvaliddate {
  my $input = shift;
  if ($input =~ m!^((?:19|20)\d\d)[- /.](0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])$!) {
    # At this point, $1 holds the year, $2 the month and $3 the day of the date entered
    if ($3 == 31 and ($2 == 4 or $2 == 6 or $2 == 9 or $2 == 11)) {
      return 0; # 31st of a month with 30 days
    } elsif ($3 >= 30 and $2 == 2) {
      return 0; # February 30th or 31st
    } elsif ($2 == 2 and $3 == 29 and not ($1 % 4 == 0 and ($1 % 100 <> 0 or $1 % 400 == 0))) {
      return 0; # February 29th outside a leap year
    } else {
      return 1; # Valid date
    }
  } else {
    return 0; # Not a date
  }
}
```

To match a date in mm/dd/yyyy format, rearrange the regular expression to «`(0[1-9]|1[012])[-/.](0[1-9]|[12][0-9]|3[01])[-/.](19|20)\d\d`» . For dd-mm-yyyy format, use «`(0[1-9]|[12][0-9]|3[01])[-/.](0[1-9]|1[012])[-/.](19|20)\d\d`».

5. Matching Whole Lines of Text

Often, you want to match complete lines in a text file rather than just the part of the line that satisfies a certain requirement. This is useful if you want to delete entire lines in a search-and-replace in a text editor, or collect entire lines in an information retrieval tool. To keep this example simple, let's say we want to match lines containing the word “John”. The regex «`John`» makes it easy enough to locate those lines. But the software will only indicate „`John`” as the match, not the entire line containing the word.

The solution is fairly simple. To specify that we need an entire line, we will use the caret and dollar sign and turn on the option to make them match at embedded newlines. In software aimed at working with text files like EditPad Pro and PowerGREP, the anchors always match at embedded newlines. To match the parts of the line before and after the match of our original regular expression «`John`», we simply use the dot and the star. Be sure to turn *off* the option for the dot to match newlines.

The resulting regex is: «`^.*John.*$`». You can use the same method to expand the match of any regular expression to an entire line, or a block of complete lines. In some cases, such as when using alternation, you will need to group the original regex together using round brackets.

Finding Lines Containing or Not Containing Certain Words

If a line can meet any out of series of requirements, simply use alternation in the regular expression. «`^.*\b(one|two|three)\b.*$`» matches a complete line of text that contains any of the words “one”, “two” or “three”. The first backreference will contain the word the line actually contains. If it contains more than one of the words, then the last (rightmost) word will be captured into the first backreference. This is because the star is greedy. If we make the first star lazy, like in «`^.*?\b(one|two|three)\b.*$`», then the backreference will contain the first (leftmost) word.

If a line must satisfy all of multiple requirements, we need to use lookahead. «`^(?=.*?\bone\b)(?=.*?\btwo\b)(?=.*?\bthree\b).*$`» matches a complete line of text that contains *all* of the words “one”, “two” and “three”. Again, the anchors must match at the start and end of a line and the dot must not match line breaks. Because of the caret, and the fact that lookahead is zero-width, all of the three lookaheads are attempted at the start of the each line. Each lookahead will match any piece of text on a single line («`.*?`») followed by one of the words. All three must match successfully for the entire regex to match. Note that instead of words like «`\bword\b`», you can put any regular expression, no matter how complex, inside the lookahead. Finally, «`.*$`» causes the regex to actually match the line, after the lookaheads have determined it meets the requirements.

If your condition is that a line should *not* contain something, use negative lookahead. «`^((?!regexp).)*$`» matches a complete line that does *not* match «`regexp`». Notice that unlike before, when using positive lookahead, I repeated both the negative lookahead and the dot together. For the positive lookahead, we only need to find one location where it can match. But the negative lookahead must be tested at each and every character position in the line. We must test that «`regexp`» fails everywhere, not just somewhere.

Finally, you can combine multiple positive and negative requirements as follows: «`^(?=.*?\bmust-have\b)(?=.*?\bmandatory\b)((?!avoid|illegal).)*$`» . When checking multiple positive requirements, the «`.*`» at the end of the regular expression full of zero-width assertions made sure that we actually matched something. Since the negative requirement must match the entire line, it is easy to replace the «`.*`» with the negative test.

6. Deleting Duplicate Lines From a File

If you have a file in which all lines are sorted (alphabetically or otherwise), you can easily delete (subsequent) duplicate lines. Simply open the file in your favorite text editor, and do a search-and-replace searching for «`^(.*)(\r?\n\1)+$`» » matches a single-line string that does not allow the quote character to appear inside the string. Using the negated character class is more efficient than using a lazy dot. «`"[^"]*"`» allows the string to span across multiple lines.

«`"[^"\\\r\n]*(\\.[^"\\\r\n]*)*"`» matches a single-line string in which the quote character can appear if it is escaped by a backslash. Though this regular expression may seem more complicated than it needs to be, it is much faster than simpler solutions which can cause a whole lot of backtracking in case a double quote appears somewhere all by itself rather than part of a string. «`"[^"\\]*(\\.[^"\\]*)*"`» allows the string to span multiple lines.

You can adapt the above regexes to match any sequence delimited by two (possibly different) characters. If we use “`b`” for the starting character, “`e`” and the end, and “`x`” as the escape character, the version without escape becomes «`b[^e\r\n]*e`», and the version with escape becomes «`b[^ex\r\n]*(x.[^ex\r\n]*)*"`».

Numbers

«`\b\d+\b`» matches a positive integer number. Do not forget the word boundaries! «`[-+]?\b\d+\b`» allows for a sign.

«`\b0[xX][0-9a-fA-F]+\b`» matches a C-style hexadecimal number.

«`((\b[0-9]+)?\.)?[0-9]+\b`» matches an integer number as well as a floating point number with optional integer part. «`(\b[0-9]+\.([0-9]+\b)?|\.[0-9]+\b)`» matches a floating point number with optional integer as well as optional fractional part, but does not match an integer number.

«`((\b[0-9]+)?\.)?\b[0-9]+([eE][-+]?[0-9]+)?\b`» matches a number in scientific notation. The mantissa can be an integer or floating point number with optional integer part. The exponent is optional.

«`\b[0-9]+(\.[0-9]+)?(e[+-]?[0-9]+)?\b`» also matches a number in scientific notation. The difference with the previous example is that if the mantissa is a floating point number, the integer part is mandatory.

If you read through the floating point number example, you will notice that the above regexes are different from what is used there. The above regexes are more stringent. They use word boundaries to exclude numbers that are part of other things like identifiers. You can prepend «`[-+]?`» to all of the above regexes to include an optional sign in the regex. I did not do so above because in programming languages, the + and - are usually considered operators rather than signs.

Reserved Words or Keywords

Matching reserved words is easy. Simply use alternation to string them together: «`\b(first|second|third|etc)\b`» Again, do not forget the word boundaries.

8. Find Two Words Near Each Other

Some search tools that use boolean operators also have a special operator called "near". Searching for “term1 near term2” finds all occurrences of term1 and term2 that occur within a certain "distance" from each other. The distance is a number of words. The actual number depends on the search tool, and is often configurable.

You can easily perform the same task with the proper regular expression.

Emulating “near” with a Regular Expression

With regular expressions you can describe almost any text pattern, including a pattern that matches two words near each other. This pattern is relatively simple, consisting of three parts: the first word, a certain number of unspecified words, and the second word. An unspecified word can be matched with the shorthand character class «`\w+`». The spaces and other characters between the words can be matched with «`\W+`» (uppercase W this time).

The complete regular expression becomes «`\bword1\W+(?:\w+\W+){1,6}word2\b`» . The quantifier «`{1,6}`» makes the regex require at least one word between “word1” and “word2”, and allow at most six words.

If the words may also occur in reverse order, we need to specify the opposite pattern as well: «`\b(?:word1\W+(?:\w+\W+){1,6}word2|word2\W+(?:\w+\W+){1,6}word1)\b`»

Part 4

Tools & Languages

1. What Is grep?

Grep is a tool that originated from the UNIX world during the 1970's. It can search through files and folders (directories in UNIX) and check which lines in those files match a given regular expression. Grep will output the filenames and the line numbers or the actual lines that matched the regular expression. All in all a very useful tool for locating information stored anywhere on your computer, even (or especially) if you do not really know where to look.

Using grep

If you type `grep regex *.txt` grep will search through all text files in the current folder. It will apply the regex to each line in the files, and print (i.e. display) each line on which a match was found. This means that grep is inherently line-based. Regex matches cannot span multiple lines.

If you like to work on the command line, the traditional grep tool will make a lot of tasks easier. All Linux distributions (except tiny floppy-based ones) install a version of grep by default, usually GNU grep. If you are using Microsoft Windows, you will need to download and install it separately. If you use Borland development tools, you already have Borland's Turbo GREP installed.

grep not only works with globbed files, but also with anything you supply on the standard input. When used with standard input, grep will print all lines it reads from standard input that match the regex. E.g.: the Linux `find` command will glob the current directory and print all file names it finds, so `find | grep regex` will print only the file names that match regex.

Grep's Regex Engine

Most versions of grep use a regex-directed engine, like the regex flavors discussed in the regex tutorial in this book . However, grep does not support all the fancy regex features that modern regex flavors support. Usually, support is limited to character classes (no shorthands), the dot, the start and end of line anchors, alternation with the vertical bar, and greedy repetition with the question mark, star and plus. Depending on the version you have, you may need to escape the question mark, plus and vertical bar to give them their special meaning. Originally, grep did not support these metacharacters. They are usually still treated as literal characters when unescaped, for backward compatibility.

An enhanced version of grep is called egrep. It uses a text-directed engine. Since neither grep nor egrep support any of the special features like backreferences, lazy repetition, or lookaround, and because grep and egrep only indicate whether a match was found on a particular line or not, this distinction does not matter, except that the text-directed engine is faster.

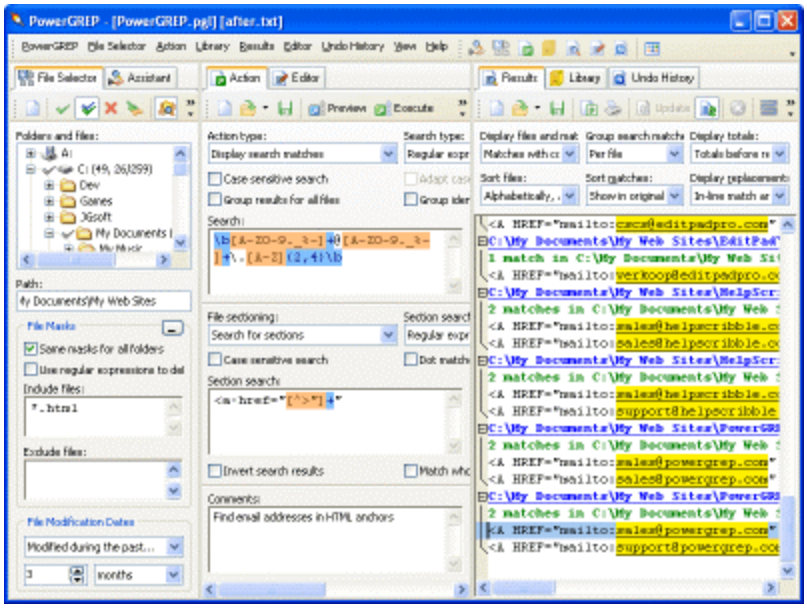
GNU grep, the most popular version of grep on Linux, uses both a text-directed and a regex-directed engine. If you use advanced features like backreferences, which GNU grep supports (but not traditional grep and egrep), it will use the regex-directed engine. Otherwise, it uses the faster text-directed engine. Again, for the tasks that grep is designed for, this does not matter to you, the user.

Beyond The Command Line

If you like to work on the command line, then the traditional grep tool is for you. But if you like to use a graphical user interface, there are many grep-like tools available for Windows and other platforms. Simply search for “grep” on your favorite software download site. Unfortunately, many grep tools come with poor documentation, leaving it up to you to figure out exactly which regex flavor they use. It's not because they claim to be Perl-compatible, that they actually are. Some are almost perfectly compatible (but never identical, though), but others fail miserably when you want to use advanced and very useful constructs like lookaround.

One Windows-based grep tool that stands out from the crowd is PowerGREP, which I will discuss next.

2. PowerGREP: Taking grep Beyond The Command Line



While all of PowerGREP's functionality is also available from the command line, the key benefit of PowerGREP over the traditional grep is its flexible and convenient graphical interface. Instead of just listing the matching lines, PowerGREP will also highlight the actual matches and make them clickable. When you click on a match, PowerGREP will load the file, with syntax coloring, allowing you to easily inspect the context of a match.

PowerGREP also provides a full-featured multi-line text editor box for composing the regular expression you want to use in your search.

PowerGREP's regex flavor supports all features of Perl 5, Java and .NET. Only the extensions that only make sense in a programming language are not available. All regex operators explained in the tutorial in this book are available in PowerGREP.

The Ultimate Search and Replace

If you already have some experience with regular expressions, then you already know that searching and replacing with regular expressions and backreferences is a powerful way to maintain all sorts of text files. If not, I suggest you download a copy of PowerGREP and take a look at the examples in the help file.

One of the benefits of using PowerGREP for such tasks, is that you can preview the replacements, and inspect the context of the replacements, just like with the search function described above. Replace or revert individual matches in PowerGREP's full-featured file editor. Naturally, an undo feature is available as well.

Another benefit is PowerGREP's ability to work with regular expression sequences. You can specify as many search and replace operations as you want, to be executed together, one after the other, on the same files. Saving sequences that you use regularly into a PowerGREP action file will save you a lot of time.

Collecting Information and Statistics

PowerGREP's “collect” feature is a unique and useful variation on the traditional regular expression search. Instead of outputting the line on which a match was found, it will output the regex match itself, or a variation of it. This variation is a piece of text you can compose using backreferences, just like the replacement text for a search and replace. You can have the collected matches sorted, and have identical matches grouped

together. This way you can compute simple statistics. The “collect” feature is most useful if you want to extract information from log files for which no specialized analysis software exists.

File Sectioning and Extra Processing

Most grep tools can work with only one regular expression at a time. With PowerGREP, you can use up to three sequences of any number of regular expressions. One sequence is the main search, search-and-replace or collect action. The other sequences are the file sectioning and extra processing. Use file sectioning to limit the main action to only certain parts of each file. Use extra processing to apply an extra search-and-replace to each individual search match.

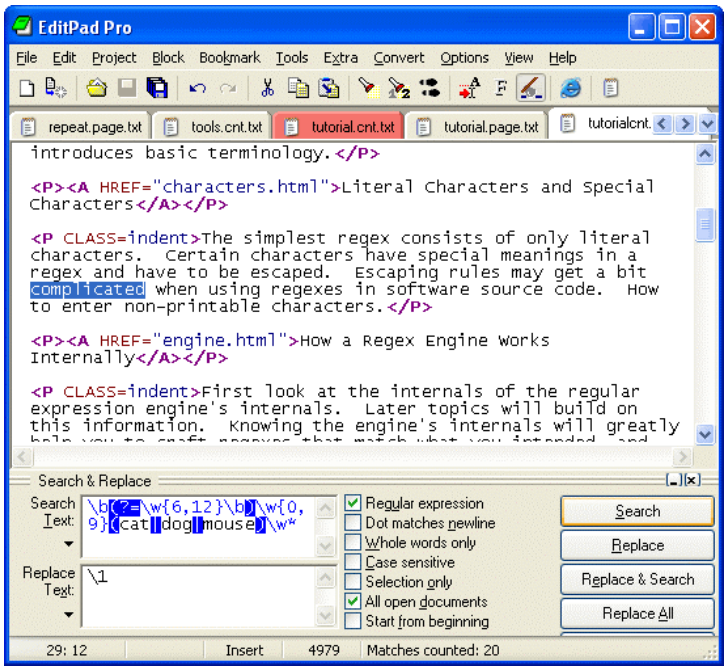
If this sounds complicated, it isn't. In fact, you'll often be able to use far simpler regular expressions with PowerGREP. E.g. instead of creating a complicated regex to match an email address inside an HTML anchor tag, use a standard regex matching an email address as the search action, and a standard regex matching an HTML anchor tag for file sectioning.

More Information on PowerGREP and Free Trial Download

PowerGREP works under Windows 95, 98, ME, NT4, 2000 and XP. For more information on PowerGREP, please visit www.powergrep.com. You can download the free evaluation version, and buy the full version for US$ 149. Not a bargain, but still very much worth its money.

3. EditPad Pro: Convenient Text Editor for Windows and Linux

EditPad Pro is one of the most convenient text editors available on the Microsoft Windows platform. You can use EditPad Pro all day long without it getting into the way of what you are trying to do. When you use search & replace and the spell checker functionality, for example, you do not get a nasty popup window blocking your view of the document you are working on, but a small, extra pane just below the text. If you often work with many files at the same time, you will save time with the tabbed interface and the Project functionality for opening and saving sets of related files.



What's more, a native Linux version of EditPad Pro is now available too. If you are moving from Windows to Linux, or using both Windows and Linux at the same time, EditPad Pro is the text editor that makes you feel at home running Linux.

EditPad Pro's Regular Expression Support

EditPad Pro's regex flavor is almost identical to the one used in Perl 5. Only the extensions that only make sense in a programming language are not available. All regex operators explained in the tutorial in this book are available in EditPad Pro.

EditPad Pro integrates with RegexBuddy. You can instantly fire up RegexBuddy to edit the regex you want to use in EditPad Pro, or select one from a RegexBuddy library.

Search and Replace Using Regular Expressions

Pressing Ctrl+F in EditPad Pro will make the search and replace pane appear. Mark the box labeled “regular expressions” to enable regex mode. Type in the regex you want to search for, mark or clear “start from beginning” and “all open documents” as you see fit, and click the Search button. EditPad Pro will then highlight the first search match, and automatically clear “start from beginning”. Click again on the Search button to continue searching for the second match, after the first. If no further matches can be found, EditPad Pro will tell you so, and automatically mark “start from beginning” so you can start all over again, if you want.

Replacing text is just as easy. First, type the replacement text, using backreferences if you want, in the Replace box. Search for the match you want to replace as above. To replace the current match, click the Replace button. To replace it and immediately search for the next match, click Replace & Search.

You can use the Replace All button to replace all search matches. Note that the Replace All button takes into account whether “start from beginning” and “all open documents” are marked. If “start from beginning” is not marked, then only matches after the text cursor will be replaced.

Custom Syntax Coloring or Highlighting Schemes

Like many modern text editors, EditPad Pro supports syntax coloring or syntax highlighting for various popular file formats and programming languages. What makes EditPad Pro unique, is that you can use regular expressions to define your own syntax coloring schemes for file types not supported by default.

To create your own coloring scheme, all you need to do is download the custom syntax coloring schemes editor (only available if you have purchased EditPad Pro), and use regular expressions to specify the different syntactic elements of the file format or programming language you want to support. The regex engine used by the syntax coloring is identical to the one used by EditPad Pro's search and replace feature, so everything you learned in the tutorial in this book applies.

The advantage is that you do not need to learn yet another scripting language or use a specific development tool to create your own syntax coloring schemes for EditPad Pro. All you need is decent knowledge of regular expressions.

More Information on EditPad Pro and Free Trial Download

EditPad Pro works under Windows 95, 98, ME, NT4, 2000 and XP. The Linux version works on all popular distributions (SuSE, Mandrake, etc.) for Intel and AMD CPUs. For more information on EditPad Pro, please visit www.editpadpro.com. You can download the free evaluation version, and buy the full version for US$ 39.95. Certainly not too expensive when you look at what you get.

4. Using Regular Expressions with Delphi for .NET and Win32

Use System.Text.RegularExpressions with Delphi for .NET

When developing Borland Delphi WinForms and VCL.NET applications, you can access all classes that are part of the Common Language Runtime (CLR), including System.Text.RegularExpressions. Simply add this namespace to the uses clause, and you can access the .NET regex classes such as Regex, Match and Group. You can use them with Delphi just as they can be used by C# and VB developers.

PCRE-based Components for Delphi for Windows/Win32

If your application is a good old Windows application using the Win32 API, you obviously cannot use the regex support from the .NET framework. Delphi itself does not provide a regular expression library, so you will need to use a third party VCL component. I recommend that you use a component that is based on the open source PCRE library. This is a very fast library, written in C. The regex syntax it supports is very complete. There are a few Delphi components that implement regular expressions purely in Delphi. Though that may sound like an advantage, the pure Delphi libraries I have seen do not support a full-featured modern regex syntax.

There are many PCRE-based VCL components available. Most are free, some are not. Some compile PCRE into a DLL that you need to ship along with your application, others link the PCRE OBJ files directly into your Delphi EXE.

One such component is TPerlRegEx, which I developed myself. You can download TPerlRegEx for free at http://www.regular-expressions.info/delphi.html. TPerlRegEx Delphi source, PCRE C sources, PCRE OBJ files and DLL are included. You can choose to link the OBJ files directly into your application, or to use the DLL. TPerlRegEx has full support for regex search-and-replace and regex splitting, which PCRE does not. Full documentation is included with the download as a help file.

RegexBuddy's Win32 Delphi code snippets are based on the TPerlRegEx component.

5. Using Regular Expressions in Java

JDK versions 1.4.0 and later have comprehensive support for regular expressions through the standard `java.util.regex`package. Because Java lacked a regex package for so long, there are also many 3rd party regex packages available for Java. I will only discuss Sun's regex library that is now part of the JDK. Its quality is excellent, better than most of the 3rd party packages. Unless you need to support older versions of the JDK, the `java.util.regex` package is the way to go.

Quick Regex Methods of The String Class

The Java String class has several methods that allow you to perform an operation using a regular expression on that string in a minimal amount of code. The downside is that you cannot specify options such as “case insensitive” or “dot matches newline”. For performance reasons, you should also not use these methods if you will be using the same regular expression often.

`myString.matches(“regex”)` returns true or false depending whether the string can be matched entirely by the regular expression. It is important to remember that String.matches() only returns true if the entire string can be matched. In other words: “regex” is applied as if you had written "^regex$" with start and end of string anchors. This is different from most other regex libraries, where the “quick match test” method returns true if the regex can be matched anywhere in the string. If myString is “`abc`” then `myString.matches(“bc”)` returns false. «`bc`» matches “`abc`”, but «`^bc$`» (which is really being used here) does not.

`myString.replaceAll(“regex”, “replacement”)` replaces all regex matches inside the string with the replacement string you specified. No surprises here. All parts of the string that match the regex are replaced. You can use the contents of capturing parentheses in the replacement text via $1, $2, $3, etc. $0 (dollar zero) inserts the entire regex match. $12 is replaced with the 12th backreference if it exists, or with the 1st backreference followed by the literal “2” if there are less than 12 backreferences. If there are 12 or more backreferences, it is not possible to insert the first backreference immediately followed by the literal “2” in the replacement text.

In the replacement text, a dollar sign not followed by a digit causes an IllegalArgumentException to be thrown. If there are less than 9 backreferences, a dollar sign followed by a digit greater than the number of backreferences throws an IndexOutOfBoundsException. So be careful if the replacement string is a user-specified string. To insert a dollar sign as literal text, use `\$` in the replacement text. When coding the replacement text as a literal string in your source code, remember that the backslash itself must be escaped too: `"\\$"`.

`myString.split(“regex”)` splits the string at each regex match. The method returns an array of strings where each element is a part of the original string between two regex matches. The matches themselves are not included in the array. Use `myString.split(“regex”, n)` to get an array containing at most n items. The result is that the string is split at most n-1 times. The last item in the string is the unsplit remainder of the original string.

Using The Pattern Class

In Java, you compile a regular expression by using the `Pattern.compile()` class factory. This factory returns an object of type `Pattern`. E.g.: `Pattern myPattern = Pattern.compile("regex");` You can specify certain options as an optional second parameter. `Pattern.compile("regex", Pattern.CASE_INSENSITIVE | Pattern.DOTALL | Pattern.MULTILINE)` makes the regex case insensitive for US ASCII characters, causes the dot to match line breaks and causes the start and end of string anchors to match at embedded line breaks as well. When working with Unicode strings, specify `Pattern.UNICODE_CASE` if you want to make the regex case insensitive for all characters in all languages. You should always specify `Pattern.CANON_EQ` to ignore differences in Unicode encodings, unless you are sure your strings contain only US ASCII characters and you want to increase performance.

If you will be using the same regular expression often in your source code, you should create a `Pattern` object to increase performance. Creating a `Pattern` object also allows you to pass matching options as a second parameter to the `Pattern.compile()` class factory. If you use one of the `String` methods above, the only way to specify options is to embed mode modifier into the regex. Putting «`(?i)`» at the start of the regex makes it case insensitive. «`(?m)`» is the equivalent of `Pattern.MULTILINE`, «`(?s)`» equals `Pattern.DOTALL` and «`(?u)`» is the same as `Pattern.UNICODE_CASE`. Unfortunately, `Pattern.CANON_EQ` does not have an embedded mode modifier equivalent.

Use `myPattern.split("subject")` to split the subject string using the compiled regular expression. This call has exactly the same results as `myString.split("regex")`. The difference is that the former is faster since the regex was already compiled.

Using The Matcher Class

Except for splitting a string (see previous paragraph), you need to create a `Matcher` object from the `Pattern` object. The `Matcher` will do the actual work. The advantage of having two separate classes is that you can create many `Matcher` objects from a single `Pattern` object, and thus apply the regular expression to many subject strings simultaneously.

To create a `Matcher` object, simply call `Pattern.matcher()` like this: `myMatcher = Pattern.matcher("subject")`. If you already created a `Matcher` object from the same pattern, call `myMatcher.reset("newsubject")` instead of creating a new matcher object, for reduced garbage and increased performance. Either way, `myMatcher` is now ready for duty.

To find the first match of the regex in the subject string, call `myMatcher.find()`. To find the next match, call `myMatcher.find()` again. When `myMatcher.find()` returns false, indicating there are no further matches, the next call to `myMatcher.find()` will find the first match again. The `Matcher` is automatically reset to the start of the string when `find()` fails.

The `Matcher` object holds the results of the last match. Call its methods `start()`, `end()` and `group()` to get details about the entire regex match and the matches between capturing parentheses. Each of these methods accepts a single int parameter indicating the number of the backreference. Omit the parameter to get information about the entire regex match. `start()` is the index of the first character in the match. `end()` is the index of the first character after the match. Both are relative to the start of the subject string. So the length of the match is `end() - start()`. `group()` returns the string matched by the regular expression or pair of capturing parentheses.

`myMatcher.replaceAll(“replacement”)` has exactly the same results as `myString.replaceAll(“regex”, “replacement”)`. Again, the difference is speed.

The `Matcher` class allows you to do a search-and-replace and compute the replacement text for each regex match in your own code. You can do this with the `appendReplacement()` and `appendTail()` Here is how:

```
StringBuffer myStringBuffer = new StringBuffer();
myMatcher = myPattern.matcher("subject");
while (myMatcher.find()) {
  if (checkIfThisMatchShouldBeReplaced()) {
    myMatcher.appendReplacement(myStringBuffer, computeReplacementString());
  }
}
myMatcher.appendTail(myStringBuffer);
```

Obviously, `checkIfThisMatchShouldBeReplaced()` and `computeReplacementString()` are placeholders for methods that you supply. The first returns true or false indicating if a replacement should be made at all. Note that skipping replacements is way faster than replacing a match with exactly the same text as was matched. `computeReplacementString()` returns the actual replacement string.

Regular Expressions, Literal Strings and Backslashes

In literal Java strings the backslash is an escape character. The literal string `"\\"` is a single backslash. In regular expressions, the backslash is also an escape character. The regular expression «\\» matches a single backslash. This regular expression as a Java string, becomes `"\\\\"`. That's right: 4 backslashes to match a single one.

The regex «\w» matches a word character. As a Java string, this is written as `"\\w"`.

The same backslash-mess occurs when providing replacement strings for methods like String.replaceAll() as literal Java strings in your Java code. In the replacement text, a dollar sign must be encoded as \$ and a backslash as \\ when you want to replace the regex match with an actual dollar sign or backslash. However, backslashes must also be escaped in literal Java strings. So a single dollar sign in the replacement text becomes `"\\$"` when written as a literal Java string. The single backslash becomes `"\\\\"`. Right again: 4 backslashes to insert a single one.

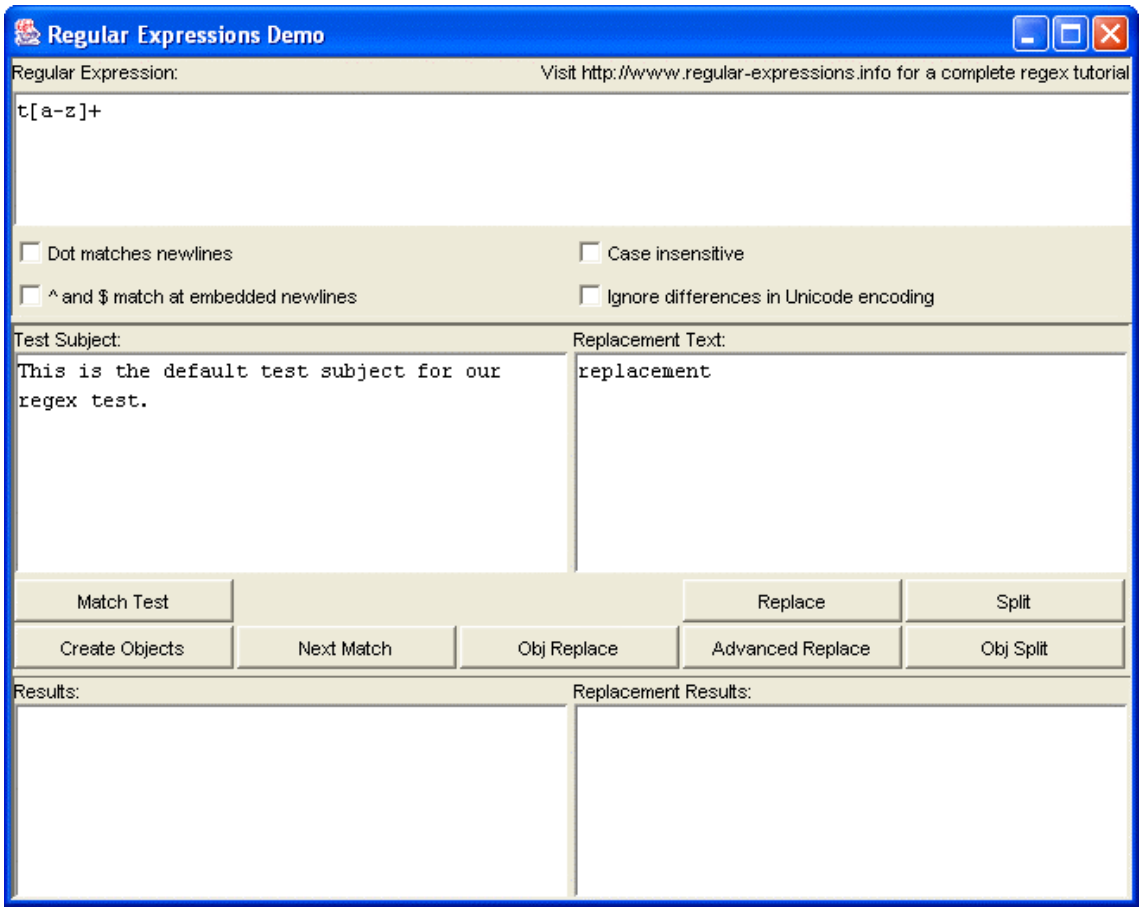
Java Demo Application using Regular Expressions

To really get to grips with the java.util.regex package, I recommend that you study the demo application I created. The demo code has lots of comments that clearly indicate what my code does, why I coded it that way, and which other options you have. The demo code also catches all exceptions that may be thrown by the various methods, something I did not explain above.

The demo application covers almost every aspect of the java.util.regex package. You can use it to learn how to use the package, and to quickly test regular expressions while coding.

6. Java Demo Application using Regular Expressions

```
package regexdemo;

import java.util.regex.*;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Regular Expressions Demo
 * Demonstration showing how to use the java.util.regex package that is part of
 * the JDK 1.4 and later
 * Copyright (c) 2003 Jan Goyvaerts.  All rights reserved.
 * Visit http://www.regular-expressions.info for a detailed tutorial
 * to regular expressions.
 * This source code is provided for educational purposes only, without any warranty of any kind.
 * Distribution of this source code and/or the application compiled
 * from this source code is prohibited.
 * Please refer everybody interested in getting a copy of the source code to
 * http://www.regular-expressions.info
 * @author Jan Goyvaerts
 * @version 1.0
 */

public class FrameRegexDemo extends JFrame {

  // Code generated by the JBuilder 9 designer to create the frame depicted below
  // has been omitted for brevity
```



```
  /** The easiest way to check if a particular string matches a regular expression
   *  is to simply call String.matches() passing the regular expression to it.
   *  It is not possible to set matching options this way, so the checkboxes
   *  in this demo are ignored when clicking btnMatch.<p>
   *
   *  One disadvantage of this method is that it will only return true if
   *  the regex matches the *entire* string.  In other words, an implicit \A
   *  is prepended to the regex and an implicit \z is appended to it.
   *  So you cannot use matches() to test if a substring anywhere in the string
   *  matches the regex.<p>
   *
   *  Note that when typing in a regular expression into textSubject,
   *  backslashes are interpreted at the regex level.
   *  Typing in \( will match a literal ( and \\ matches a literal backslash.
   *  When passing literal strings in your source code, you need to escape
   *  backslashes in strings as usual.
   *  The string "\\(" matches a literal ( character
   *  and "\\\\" matches a single literal backslash.
   */
  void btnMatch_actionPerformed(ActionEvent e) {
    textReplaceResults.setText("n/a");
    // Calling the Pattern.matches static method is an alternative way
    // if (Pattern.matches(textRegex.getText(), textSubject.getText())) {
    try {
      if (textSubject.getText().matches(textRegex.getText())) {
        textResults.setText("The regex matches the entire subject");
      }
      else {
        textResults.setText("The regex does not match the entire subject");
      }
    } catch (PatternSyntaxException ex) {
      textResults.setText("You have an error in your regular expression:\n" +
                          ex.getDescription());
    }
  }

  /** The easiest way to perform a regex search-and-replace on a string
   *  is to call the string's replaceFirst() and replaceAll() methods.
   *  replaceAll() will replace all substrings that match the regular expression
   *  with the replacement string, while replaceFirst() will only replace
   *  the first match.<p>
   *
   *  Again, you cannot set matching options this way, so the checkboxes
   *  in this demo are ignored when clicking btnMatch.<p>
   *
   *  In the replacement text, you can use $0 to insert the entire regex match,
   *  and $1, $2, $3, etc. for the backreferences (text matched by the part in the
   *  regex between the first, second, third, etc. pair of round brackets)<br>
   *  \$ inserts a single $ character.<p>
   *
   *  $$ or other improper use of the $ sign throws an IllegalArgumentException.
   *  If you reference a group that does not exist (e.g. $4 if there are only
   *  3 groups), throws an IndexOutOfBoundsException.
   *  Be sure to properly handle these exceptions if you allow the end user
   *  to type in the replacement text.<p>
   *
   *  Note that in the memo control, you type \$ to insert a dollar sign,
   *  and \\ to insert a backslash.  If you provide the replacement string as a
   *  string literal in your Java code, you need to use "\\$" and "\\\\".
   *  This is because backslashes need to be escaped in Java string literals too.
   */
  void btnReplace_actionPerformed(ActionEvent e) {
    try {
      textReplaceResults.setText(
        textSubject.getText().replaceAll(
          textRegex.getText(), textReplace.getText())
      );
      textResults.setText("n/a");
    } catch (PatternSyntaxException ex) {
      // textRegex does not contain a valid regular expression
      textResults.setText("You have an error in your regular expression:\n" +
```

```
                        ex.getDescription());
      textReplaceResults.setText("n/a");
    } catch (IllegalArgumentException ex) {
      // textReplace contains inapropriate dollar signs
      textResults.setText("You have an error in the replacement text:\n" +
                        ex.getMessage());
      textReplaceResults.setText("n/a");
    } catch (IndexOutOfBoundsException ex) {
      // textReplace contains a backreference that does not exist
      // (e.g. $4 if there are only three groups)
      textResults.setText("Non-existent group in the replacement text:\n" +
                        ex.getMessage());
      textReplaceResults.setText("n/a");
    }
  }

  /** Show the results of splitting a string. */
  void printSplitArray(String[] array) {
    textResults.setText(null);
    for (int i = 0; i < array.length; i++) {
      textResults.append(Integer.toString(i) + ": \"" + array[i] + "\"\r\n");
    }
  }

  /** The easiest way to split a string into an array of strings is by calling
   *  the string's split() method.  The string will be split at each substring
   *  that matches the regular expression.  The regex matches themselves are
   *  thrown away.<p>
   *
   *  If the split would result in trailing empty strings, (when the regex matches
   *  at the end of the string), the trailing empty strings are also thrown away.
   *  If you want to keep the empty strings, call split(regex, -1).  The -1 tells
   *  the split() method to add trailing empty strings to the resulting array.<p>
   *
   *  You can limit the number of items in the resulting array by specifying a
   *  positive number as the second parameter to split().  The limit you specify
   *  it the number of items the array will at most contain.  The regex is applied
   *  at most limit-1 times, and the last item in the array contains the unsplit
   *  remainder of the original string.  If you are only interested in the first
   *  3 items in the array, specify a limit of 4 and disregard the last item.
   *  This is more efficient than having the string split completely.
   */
  void btnSplit_actionPerformed(ActionEvent e) {
    textReplaceResults.setText("n/a");
    try {
      printSplitArray(textSubject.getText().split(textRegex.getText()
                                                  /*, Limit*/ ));
    } catch (PatternSyntaxException ex) {
      // textRegex does not contain a valid regular expression
      textResults.setText("You have an error in your regular expression:\n" +
                        ex.getDescription());
    }
  }

  /** Figure out the regex options to be passed to the Pattern.compile()
   *  class factory based on the state of the checkboxes.
   */
  int getRegexOptions() {
    int Options = 0;
    if (checkCanonEquivalence.isSelected()) {
      // In Unicode, certain characters can be encoded in more than one way.
      // Many letters with diacritics can be encoded as a single character
      // identifying the letter with the diacritic, and encoded as two
      // characters: the letter by itself followed by the diacritic by itself
      // Though the internal representation is different, when the string is
      // rendered to the screen, the result is exactly the same.
      Options |= Pattern.CANON_EQ;
    }
    if (checkCaseInsensitive.isSelected()) {
      // Omitting UNICODE_CASE causes only US ASCII characters to be matched
      // case insensitively.  This is appropriate if you know beforehand that
```

```
      // the subject string will only contain US ASCII characters
      // as it speeds up the pattern matching.
      Options |= Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE;
    }
    if (checkDotAll.isSelected()) {
      // By default, the dot will not match line break characters.
      // Specify this option to make the dot match all characters,
      // including line breaks
      Options |= Pattern.DOTALL;
    }
    if (checkMultiLine.isSelected()) {
      // By default, the caret ^, dollar $  only match at the start
      // and the end of the string.  Specify this option to make ^ also match
      // after line breaks in the string, and make $ match before line breaks.
      Options |= Pattern.MULTILINE;
    }
    return Options;
  }

  /** Pattern constructed by btnObject */
  Pattern compiledRegex;

  /** Matcher object that will search the subject string using compiledRegex */
  Matcher regexMatcher;
  JLabel jLabel8 = new JLabel();
  JButton btnAdvancedReplace = new JButton();

  /** If you will be using a particular regular expression often,
   *  you should create a Pattern object to store the regular expression.
   *  You can then reuse the regex as often as you want by reusing the
   *  Pattern object.<p>
   *
   *  To use the regular expression on a string, create a Matcher object
   *  by calling compiledRegex.matcher() passing the subject string to it.
   *  The Matcher will do the actual searching, replacing or splitting.<p>
   *
   *  You can create as many Matcher objects from a single Pattern object
   *  as you want, and use the Matchers at the same time.  To apply the regex
   *  to another subject string, either create a new Matcher using
   *  compiledRegex.matcher() or tell the existing Matcher to work on a new
   *  string by calling regexMatcher.reset(subjectString).
   */
  void btnObjects_actionPerformed(ActionEvent e) {
    compiledRegex = null;
    textReplaceResults.setText("n/a");
    try {
      // If you do not want to specify any options (this is the case when
      // all checkboxes in this demo are unchecked), you can omit the
      // second parameter for the Pattern.compile() class factory.
      compiledRegex = Pattern.compile(textRegex.getText(), getRegexOptions());
      // Create the object that will search the subject string
      // using the regular expression.
      regexMatcher = compiledRegex.matcher(textSubject.getText());
      textResults.setText("Pattern and Matcher objects created.");
    } catch (PatternSyntaxException ex) {
      // textRegex does not contain a valid regular expression
      textResults.setText("You have an error in your regular expression:\n" +
                          ex.getDescription());
    } catch (IllegalArgumentException ex) {
      // This exception indicates a bug in getRegexOptions
      textResults.setText("Undefined bit values are set in the regex options");
    }
  }

  /** Print the results of a search produced by regexMatcher.find()
   *  and stored in regexMatcher.
   */
  void printMatch() {
    try {
      textResults.setText("Index of the first character in the match: " +
                          Integer.toString(regexMatcher.start()) + "\n");
```

```
    textResults.append("Index of the first character after the match: " +
                       Integer.toString(regexMatcher.end()) + "\n");
    textResults.append("Length of the match: " +
                       Integer.toString(regexMatcher.end() -
                                        regexMatcher.start()) + "\n");
    textResults.append("Matched text: " + regexMatcher.group() + "\n");
    if (regexMatcher.groupCount() > 0) {
      // Capturing parentheses are numbered 1..groupCount()
      // group number zero is the entire regex match
      for (int i = 1; i <= regexMatcher.groupCount(); i++) {
        String groupLabel = new String("Group " + Integer.toString(i));
        if (regexMatcher.start(i) < 0) {
          textResults.append(groupLabel +
                             " did not participate in the overall match\n");
        } else {
          textResults.append(groupLabel + " start: " +
                             Integer.toString(regexMatcher.start(i)) + "\n");
          textResults.append(groupLabel + " end: " +
                             Integer.toString(regexMatcher.end(i)) + "\n");
          textResults.append(groupLabel + " length: " +
                             Integer.toString(regexMatcher.end(i) -
                                              regexMatcher.start(i)) + "\n");
          textResults.append(groupLabel + " matched text: " +
                             regexMatcher.group(i) + "\n");
        }
      }
    }
  } catch (IllegalStateException ex) {
    // Querying the results of a Matcher object before calling find()
    // or after a call to find() returned False, throws an IllegalStateException
    // This indicates a bug in our application
    textResults.setText("Cannot print match results if there aren't any");
  } catch (IndexOutOfBoundsException ex) {
    // Querying the results of groups (capturing parentheses or backreferences)
    // that do not exist throws an IndexOutOfBoundsException
    // This indicates a bug in our application
    textResults.setText("Cannot print match results of non-existent groups");
  }
}

/** Finds the first match if this is the first search, or if the previous search
 *  came up empty. Otherwise, it finds the next match after the previous match.
 *
 *  Note that even if you typed in new text for the regex or subject,
 *  btnNextMatch uses the subject and regex as they were when you clicked
 *  btnCreateObjects.
 */
void btnNextMatch_actionPerformed(ActionEvent e) {
  textReplaceResults.setText("n/a");
  if (regexMatcher == null) {
    textResults.setText("Click Create Objects to create the Matcher object");
  } else {
    // Caling Matcher.find() without any parameters continues the search at
    // Matcher.end().  Starts from the beginning of the string if this is
    // the first search using the Matcher or if the previous search
    // did not find any (further) matches.
    if (regexMatcher.find()) {
      printMatch();
    } else {
      // This also resets the starting position for find()
      // to the start of the subject string
      textResults.setText("No further matches");
    }
  }
}

/** Perform a regular expression search-and-replace using a Matcher object.
 *  This is the recommended way if you often use the same regular expression
 *  to do a search-and-replace.  You should also reuse the Matcher object
 *  by calling Matcher.reset(nextSubjectString) for improved efficiency.<p>
 *
```

```
   *  You also need to use the Pattern and Matcher objects for the
   *  search-and-replace if you want to use the regex options such as
   *  "case insensitive" or "dot all".<p>
   *
   *  See the btnReplace notes for the special $-syntax in the replacement text.
   */
  void btnObjReplace_actionPerformed(ActionEvent e) {
    if (regexMatcher == null) {
      textResults.setText("Click Create Objects to create the Matcher object");
    } else {
      try {
        textReplaceResults.setText(regexMatcher.replaceAll(textReplace.getText()));
      } catch (IllegalArgumentException ex) {
        // textReplace contains inapropriate dollar signs
        textResults.setText("You have an error in the replacement text:\n" +
                            ex.getMessage());
        textReplaceResults.setText("n/a");
      } catch (IndexOutOfBoundsException ex) {
        // textReplace contains a backreference that does not exist
        // (e.g. $4 if there are only three groups)
        textResults.setText("Non-existent group in the replacement text:\n" +
                            ex.getMessage());
        textReplaceResults.setText("n/a");
      }
    }
  }

  /** Using Matcher.appendReplacement() and Matcher.appendTail() you can implement
   *  a search-and-replace of arbitrary complexity.  These routines allow you
   *  to compute the replacement string in your own code.  So the replacement text
   *  can be whatever you want.<p>
   *
   *  To do this, simply call Matcher.find() in a loop.  For each match returned
   *  by find(), call appendReplacement() with whatever replacement text you want.
   *  When find() can no longer find matches, call appendTail().<p>
   *
   *  appendReplacement() appends the substring between the end of the previous
   *  match that was replaced with appendReplacement() and the current match.
   *  If this is the first call to appendReplacement() since creating the Matcher
   *  or calling reset(), then the appended substring starts at the start of
   *  the string.  Then, the specified replacement text is appended.
   *  If the replacement text contains dollar signs, they will be interpreted
   *  as usual.  E.g. $1 is replaced with the match between the first pair of
   *  capturing parentheses.<p>
   *
   *  appendTail() appends the substring between the end of the previous match
   *  that was replaceced with appendReplacement() and the end of the string.
   *  If appendReplacement() was not called since creating the Matcher or
   *  calling reset(), the entire subject string is appended.<p>
   *
   *  The above means that you should call Matcher.reset() before starting the
   *  operation, unless you're sure the Matcher is freshly constructed.
   *  If certain matches do not need to be replaced, simply skip calling
   *  appendReplacement() for those matches. (Calling appendReplacement() with
   *  Matcher.group() as the replacement text will only hurt performance and
   *  may get you into trouble with dollar signs that may appear in the match.)
   */
  void btnAdvancedReplace_actionPerformed(ActionEvent e) {
    if (regexMatcher == null) {
      textResults.setText("Click Create Objects to create the Matcher object");
    } else {
      // We will store the replacement text here
      StringBuffer replaceResult = new StringBuffer();
      while (regexMatcher.find()) {
        try {
          // In this example, we simply replace the regex match with the same text
          // in uppercase.  Note that appendReplacement parses the replacement
          // text to substitute $1, $2, etc. with the contents of the
          // corresponding capturing parentheses just like replaceAll()
          regexMatcher.appendReplacement(replaceResult,
                                         regexMatcher.group().toUpperCase());
```

```
        } catch (IllegalStateException ex) {
          // appendReplacement() was called without a prior successful call to find()
          // This exception indicates a bug in your source code
          textResults.setText("appendReplacement() called without a prior" +
                              "successful call to find()");
          textReplaceResults.setText("n/a");
          return;
        } catch (IllegalArgumentException ex) {
          // Replacement text contains inapropriate dollar signs
          textResults.setText("Error in the replacement text:\n" +
                              ex.getMessage());
          textReplaceResults.setText("n/a");
          return;
        } catch (IndexOutOfBoundsException ex) {
          // Replacement text contains a backreference that does not exist
          // (e.g. $4 if there are only three groups)
          textResults.setText("Non-existent group in the replacement text:\n" +
                              ex.getMessage());
          textReplaceResults.setText("n/a");
          return;
        }
      }
      regexMatcher.appendTail(replaceResult);
      textReplaceResults.setText(replaceResult.toString());
      textResults.setText("n/a");
      // After using appendReplacement and appendTail, the Matcher object must be
      // reset so we can use appendReplacement and appendTail again.
      // In practice, you will probably put this call at the start of the routine
      // where you want to use appendReplacement and appendTail.
      // I did not do that here because this way you can click on the Next Match
      // button a couple of times to skip a few matches, and then click on the
      // Advanced Replace button to observe that appendReplace() will copy the
      // skipped matches unchanged.
      regexMatcher.reset();
    }
  }

  /** If you want to split many strings using the same regular expression,
   *  you should create a Pattern object and call Pattern.split()
   *  rather than String.split().  Both methods produce exactly the same results.
   *  However, when creating a Pattern object, you can specify options such as
   *  "case insensitive" and "dot all".<p>
   *
   *  Note that no Matcher object is used.
   */
  void btnObjSplit_actionPerformed(ActionEvent e) {
    textReplaceResults.setText("n/a");
    if (compiledRegex == null) {
      textResults.setText("Please click Create Objects to compile the regex");
    } else {
      printSplitArray(compiledRegex.split(textSubject.getText() /*, Limit*/));
    }
  }
}

// ActionListener classes generated by JBuilder 9 have been omitted for brevity
```

7. Using Regular Expressions with JavaScript and ECMAScript

JavaScript 1.2 and later (also known as, JScript ECMAScript or ECMA-262) has built-in support for regular expressions. MSIE 4 and later, Netscape 4 and later, all versions of Firefox, and most other modern web browsers support JavaScript 1.2. If you use JavaScript to validate user input on a web page at the client side, using JavaScript's regular expression support will greatly reduce the amount of code you need to write.

In JavaScript, a regular expression is written in the form of `/pattern/modifiers` where “pattern” is the regular expression itself, and “modifiers” are a series of characters indicating various options. The “modifiers” part is optional. This syntax is borrowed from Perl. JavaScript supports the following modifiers, a subset of those supported by Perl:

- `/g` enables "global" matching. When using the `replace()` method, specify this modifier to replace all matches, rather than only the first one.
- `/i` makes the regex match case insensitive.
- `/m` enables "multi-line mode". In this mode, the caret and dollar match before and after newlines in the subject string.

You can combine multiple modifiers by stringing them together as in `/regex/gim`. Notably absent is an option to make the dot match line break characters.

Since forward slashes delimit the regular expression, any forward slashes that appear in the regex need to be escaped. E.g. the regex «`1/2`» is written as `/1\/2/` in JavaScript.

Regexp Methods of The String Class

To test if a particular regex matches (part of) a string, you can call the strings's `match()` method: `if (myString.match(/regex/) { /*Success!*/ }`. If you want to verify user input, you should use anchors to make sure that you are testing against the entire string. To test if the user entered a number, use: `myString.match(/^\d+$/)`. `/\d+/` matches any string containing one or more digits, but `/^\d+$/` matches only strings consisting entirely of digits.

To do a search and replace with regexes, use the string's `replace()` method: `myString.replace(/replaceme/g, “replacement”)`. Using the `/g` modifier makes sure that all occurrences of “replaceme” are replaced. The second parameter is an normal string with the replacement text.

If the regexp contains capturing parentheses, you can use backreferences in the replacement text. `$1` in the replacement text inserts the text matched by the first capturing group, `$2` the second, etc. up to `$9`.

Finally, using a string's `split()` method allows you to split the string into an array of strings using a regular expression to determine the positions at which the string is splitted. E.g. `myArray = myString.split(/,/)` splits a comma-delimited list into an array. The comma's themselves are not included in the resulting array of strings.

How to Use The JavaScript RegExp Object

Each JavaScript execution thread (i.e. each browser window or frame) contains one pre-initialized RegExp object. Usually, you will not use this object directly. The easiest way to create a new regexp instance is to simply use the special regex syntax: `myregexp = /regex/`.

If you have the regular expression in a string (e.g. because it was typed in by the user), you can use the RegExp constructor: `myregexp = new RegExp(regexstring)`. Modifiers can be specified as a second parameter: `myregexp = new RegExp(regexstring, “gims”)`.

I recommend that you do not use the RegExp constructor with a literal string, because in literal strings, backslashes must be escaped. The regular expression «\w+» can be created as `re = /\w+/` or as `re = new RegExp("\\w+")`. The latter is definitely harder to read. The regular expression «\\» matches a single backslash. In JavaScript, this becomes `re = /\\/` or `re = new RegExp("\\\\")`.

Whichever way you create “myregexp”, you can pass it to the String methods explained above instead of a literal regular expression: `myString.replace(myregexp, “replacement”)`.

If you want to retrieve the part of the string that was matched, call the `exec()` function of the RegExp object that you created, e.g.: `mymatch = myregexp.exec(“subject”)`. This function returns an array. The zeroth item in the array will hold the text that was matched by the regular expression. The following items contain the text matched by the capturing parentheses in the regexp, if any. `mymatch.index` indicates the character position in the subject string at which the pattern matched.

Calling the `exec()` function also changes a number of properties of the RegExp object. Note that even though you can create multiple “myregexp” instances, each JavaScript thread of execution only has one global RegExp object. This means that the property values of all the “myregexp” instances will all be the same, and indicate the result of the very last call to `exec()`. The `lastMatch` property holds the text matched by the last call to `exec()`, and `lastIndex` stores the index in the subject string of the first character in the match. `leftContext` stores the part of the subject string to the left or the regexp match, and `rightContext` the part to the right.

8. Using Regular Expressions with The Microsoft .NET Framework

The Microsoft .NET Framework, which you can use with any .NET programming language such as C# (C sharp) or Visual Basic.NET, has solid support for regular expressions. The documentation of the regular expression classes is very poor, however. Read on to learn how to use regular expressions in your .NET applications. In the text below, I will use VB.NET syntax to explain the various classes. After the text, you will find a complete application written in C# to illustrate how to use regular expressions in great detail. I recommed that you download the source code, read the source code and play with the application. That will give you a clear idea how to use regexes in your own applications.

System.Text.RegularExpressions Overview (Using VB.NET Syntax)

The regex classes are located in the namespace System.Text.RegularExpressions. To make them available, place `Imports System.Text.RegularExpressions` at the start of your source code.

The `Regex` class is the one you use to compile a regular expression. For efficiency, regular expressions are compiled into an internal format. If you plan to use the same regular expression repeatedly, construct a `Regex` object as follows: `Dim RegexObj as Regex = New Regex("regularexpression")`. You can then call `RegexObj.IsMatch("subject")` to check whether the regular expression matches the subject string. The `Regex` allows an optional second parameter of type `RegexOptions`. You could specify `RegexOptions.IgnoreCase` as the final parameter to make the regex case insensitive. Other options are `RegexOptions.Singleline` which causes the dot to match newlines and `RegexOptions.Multiline` which causes the caret and dollar to match at embedded newlines in the subject string.

Call `RegexObj.Replace("subject", "replacement")` to perform a search-and-replace using the regex on the subject string, replacing all matches with the replacement string. In the replacement string, you can use `$&` to insert the entire regex match into the replacement text. You can use `$1, $2, $3, etc...` to insert the text matched between capturing parentheses into the replacement text. Use `$$` to insert a single dollar sign into the replacement text. To replace with the first backreference immediately followed by the digit 9, use `${1}9`. If you type $19, and there are less than 19 backreferences, the `$19` will be interpreted as literal text, and appear in the result string as such. To insert the text from a named capturing group, use `${name}`. Improper use of the $ sign may produce an undesirable result string, but will never cause an exception to be raised.

`RegexObj.Split("Subject")` splits the subject string along regex matches, returning an array of strings. The array contains the text between the regex matches. If the regex contains capturing parentheses, the text matched by them is also included in the array. If you want the entire regex matches to be included in the array, simply place round brackets around the entire regular expression when instantiating `RegexObj`.

The `Regex` class also contains several static methods that allow you to use regular expressions without instantiating a `Regex` object. This reduces the amount of code you have to write, and is appropriate if the same regular expression is used only once or reused seldomly. Note that member overloading is used a lot in the `Regex` class. All the static methods have the same names (but different parameter lists) as other non-static methods.

`Regex.IsMatch("subject", "regex")` checks if the regular expression matches the subject string. `Regex.Replace("subject", "regex", "replacement")` performs a search-and-replace.

`Regex.Split("subject", "regex")` splits the subject string into an array of strings as described above. All these methods accept an optional additional parameter of type `RegexOptions`, like the constructor.

The System.Text.RegularExpressions.Match Class

If you want more information about the regex match, call `Regex.Match()` to construct a `Match` object. If you instantiated a `Regex` object, use `Dim MatchObj as Match = RegexObj.Match("subject")`. If not, use the static version: `Dim MatchObj as Match = Regex.Match("subject", "regex")`.

Either way, you will get an object of class `Match` that holds the details about the first regex match in the subject string. `MatchObj.Success` indicates if there actually was a match. If so, use `MatchObj.Value` to get the contents of the match, `MatchObj.Length` for the length of the match, and `MatchObj.Index` for the start of the match in the subject string. The start of the match is zero-based, so it effectively counts the number of characters in the subject string to the left of the match.

If the regular expression contains capturing parentheses, use the `MatchObj.Groups` collection. `MatchObj.Groups.Count` indicates the number of capturing parentheses. The count includes the zeroth group, which is the entire regex match. `MatchObj.Groups(3).Value` gets the text matched by the third pair of round brackets. `MatchObj.Groups(3).Length` and `MatchObj.Groups(3).Index` get the length of the text matched by the group and its index in the subject string, relative to the start of the subject string. `MatchObj.Groups("name")` gets the details of the named group "name".

To find the next match of the regular expression in the same subject string, call `MatchObj.NextMatch()` which returns a new `Match` object containing the results for the second match attempt. You can continue calling `MatchObj.NextMatch()` until `MatchObj.Success` is False.

Note that after calling `RegexObj.Match()`, the resulting `Match` object is independent from `RegexObj`. This means you can work with several `Match` objects created by the same `Regex` object simultaneously.

Regular Expressions, Literal Strings and Backslashes

In literal C# strings, as well as in C++ and many other .NET languages, the backslash is an escape character. The literal string `"\\"` is a single backslash. In regular expressions, the backslash is also an escape character. The regular expression «\\» matches a single backslash. This regular expression as a C# string, becomes `"\\\\"`. That's right: 4 backslashes to match a single one.

The regex «\w» matches a word character. As a C# string, this is written as `"\\w"`.

To make your code more readable, you should use C# verbatim strings. In a verbatim string, a backslash is an ordinary character. This allows you to write the regular expression in your C# code as you would write it a tool like RegexBuddy or PowerGREP, or as the user would type it into your application. The regex to match a backlash is written as `@"\\"` when using C# verbatim strings. The backslash is still an escape character in the regular expression, so you still need to double it. But doubling is better than quadrupling. To match a word character, use the verbatim string `@"\w"`.

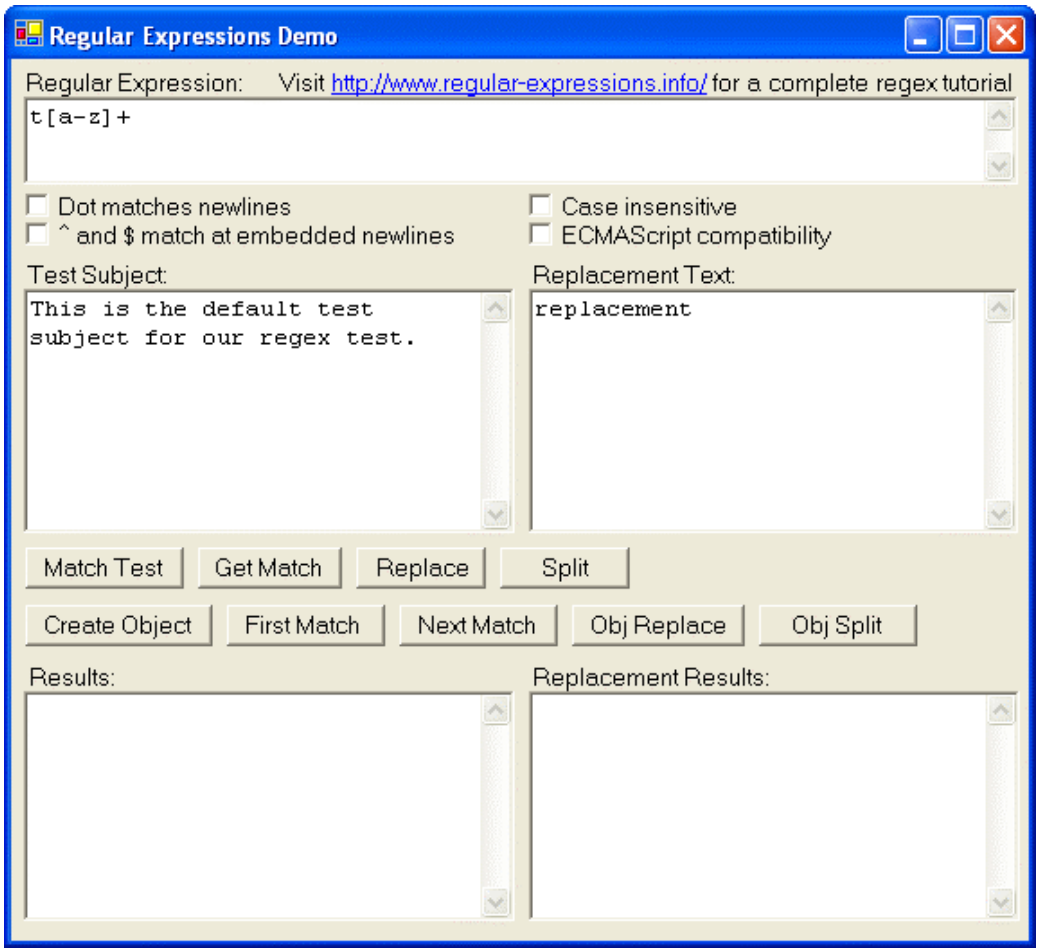
.NET Framework Demo Application using Regular Expressions (C# Syntax)

To really get to grips with the regex support of the Microsoft .NET Framework, I recommend that you study the demo application I created. It is written in C#. The demo is fairly simple, so you should understand the source code even if you do not use C# yourself. The demo code has lots of comments that clearly indicate what my code does, why I coded it that way, and which other options you have. The demo code also catches all exceptions that may be thrown by the various methods, something I did not explain above.

The demo application covers every aspect of the System.Text.RegularExpressions package. You can use it to learn how to use the package, and to quickly test regular expressions while coding.

9. C# Demo Application

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

// This line allows us to use classes like Regex and Match
// without having to spell out the entire location.
using System.Text.RegularExpressions;

namespace RegexDemo
{
  /// <summary>
  /// Application showing the use of regular expressions in the .NET framework
  /// Copyright (c) 2003 Jan Goyvaerts.  All rights reserved.
  /// Visit http://www.regular-expressions.info for a detailed tutorial to regular expressions.
  ///
  /// This source code is provided for educational purposes only, without
  /// any warranty of any kind.  Distribution of this source code and/or the
  /// application compiled from this source code is prohibited.  Please refer
  /// everybody interested in getting a copy of the source code to
  /// http://www.regular-expressions.info where it can be downloaded.
  /// </summary>
  public class FormRegex : System.Windows.Forms.Form
  {
    // Designer-generated code to create the form has been omitted for brevity
```



```
private void checkDotAll_Click(object sender, System.EventArgs e)
{
  // "Dot all" and "ECMAScript" are mutually exclusive options.
  if (checkDotAll.Checked) checkECMAScript.Checked = false;
}

private void checkECMAScript_Click(object sender, System.EventArgs e)
{
  // "Dot all" and "ECMAScript" are mutually exclusive options.
  if (checkECMAScript.Checked) checkDotAll.Checked = false;
}

private RegexOptions getRegexOptions()
{
  // "Dot all" and "ECMAScript" are mutually exclusive options.
  // If we include them both, then the Regex() constructor or the
  // Regex.Match() method will raise an exception
  System.Diagnostics.Trace.Assert(
         !(checkDotAll.Checked && checkECMAScript.Checked),
         "DotAll and ECMAScript options are mutually exclusive");
  // Construct a RegexOptions object
  // If the options are predetermined, you can simply pass something like
  // RegexOptions.Multiline | RegexOptions.Ignorecase
  // directly to the Regex() constructor or the Regex.Match() method
  RegexOptions options = new RegexOptions();
  // If true, the dot matches any character, including a newline
  // If false, the dot matches any character, except a newline
  if (checkDotAll.Checked) options |= RegexOptions.Singleline;
  // If true, the caret ^ matches after a newline, and the dollar $ matches
  // before a newline, as well as at the start and end of the subject string
  // If false, the caret only matches at the start of the string
  // and the dollar only at the end of the string
  if (checkMultiLine.Checked) options |= RegexOptions.Multiline;
  // If true, the regex is matched case insensitively
  if (checkIgnoreCase.Checked) options |= RegexOptions.IgnoreCase;
  // If true, \w, \d and \s match ASCII characters only,
  // and \10 is backreference 1 followed by a literal 0
  // rather than octal escape 10.
  if (checkECMAScript.Checked) options |= RegexOptions.ECMAScript;
  return options;
}

private void btnMatch_Click(object sender, System.EventArgs e)
{
  // This method illustrates the easiest way to test if a string can be
  // matched by a regex using the System.Text.RegularExpressions.Regex.Match
  // static method.  This way is recommended when you only want to validate
  // a single string every now and then.

  // Note that IsMatch() will also return True if the regex matches part of
  // the string only.  If you only want it to return True if the regex matches
  // the entire string, simply prepend a caret and append a dollar sign
  // to the regex to anchor it at the start and end.

  // Note that when typing in a regular expression into textSubject,
  // backslashes are interpreted at the regex level.
  // So typing in \( will match a literal ( character and \\ matches a
  // literal backslash.  When passing literal strings in your source code,
  // you need to escape backslashes in strings as usual.
  // So the string "\\(" matches a literal ( and "\\\\" matches a single
  // literal backslash.
  // To reduce confusion, I suggest you use verbatim strings instead:
  // @"\(" matches a literal ( and @"\\" matches a literal backslash.
  // You can omit the last parameter with the regex options
  // if you don't want to specify any.
  textReplaceResults.Text = "N/A";
  try
  {
    if (Regex.IsMatch(textSubject.Text, textRegex.Text, getRegexOptions())) {
      textResults.Text = "The regex matches part or all of the subject";
    } else {
```

```
      textResults.Text = "The regex cannot be matched in the subject";
    }
  }
  catch (Exception ex)
  {
    // Most likely cause is a syntax error in the regular expression
    textResults.Text = "Regex.IsMatch() threw an exception:\r\n" + ex.Message;
  }
}

private void btnGetMatch_Click(object sender, System.EventArgs e)
{
  // Illustrates the easiest way to get the text of the first match
  // using the System.Text.RegularExpressions.Regex.Match static method.
  // Useful for easily extracting a string form another string.
  // You can omit the last parameter with the regex options
  // if you don't want to specify any.
  // If there's no match, Regex.Match.Value returns an empty string.
  // If you are only interested in part of the regex match, you can use
  // .Groups[3].Value instead of .Value to get the text matched between
  // the third pair of round brackets in the regular expression
  textReplaceResults.Text = "N/A";
  try
  {
    textResults.Text = Regex.Match(textSubject.Text, textRegex.Text,
                                   getRegexOptions()).Value;
  }
  catch (Exception ex)
  {
    // Most likely cause is a syntax error in the regular expression
    textResults.Text = "Regex.Match() threw an exception:\r\n" + ex.Message;
  }
}

private void btnReplace_Click(object sender, System.EventArgs e)
{
  // Illustrates the easiest way to do a regex-based search-and-replace on
  // a single string using the System.Text.RegularExpressions.Regex.Replace
  // static method.  This method will replace ALL matches of the regex in
  // the subject with the replacement text.
  // If there are no matches, Replace() returns the subject string unchanged.
  // If you only want to replace certain matches, you have to use the method
  // illustrated in btnRegexObjReplace_click.
  // You can omit the last parameter with the regex options
  // if you don't want to specify any.
  // In the replacement text (textReplace.Text), you can use $& to insert
  // the entire regex match, and $1, $2, $3, etc. for the backreferences
  // (text matched by the part in the regex between the first, second,
  //  third, etc. pair of round brackets)
  // $$ inserts a single $ character
  // $` (dollar backtick) inserts the text in the subject
  //     to the left of the regex match
  // $' (dollar single quote) inserts the text in the subject
  //     to the right of the end of the regex match
  // $_ inserts the entire subject text
  try
  {
    textReplaceResults.Text = Regex.Replace(textSubject.Text, textRegex.Text,
                              textReplace.Text, getRegexOptions());
    textResults.Text = "N/A";
  }
  catch (Exception ex)
  {
    // Most likely cause is a syntax error in the regular expression
    textResults.Text = "Regex.Replace() threw an exception:\r\n" + ex.Message;
    textReplaceResults.Text = "N/A";
  }
}

private void printSplitArray(string[] array) {
  textResults.Text = "";
```

```
    for (int i = 0; i < array.Length; i++) {
      textResults.AppendText(i.ToString() + ": \"" + array[i] + "\"\r\n");
    }
  }

  private void btnSplit_Click(object sender, System.EventArgs e)
  {
    // Regex.Split allows you to split a single string into an array of strings
    // using a regular expression.  This example illustrates the easiest way
    // to do this; use btnRegexObjSplit_Click if you need to split many strings.
    // The string is cut at each point where the regex matches.  The part of
    // the string matched by the regex is thrown away.  If the regex contains
    // capturing parentheses, then the part of the string matched by each of
    // them is also inserted into the array.
    // To summarize, the array will contain
    // (indenting for clarity; the array is one-dimensional):
    // - the part of the string before the first regex match
    //   - the part of the string captured in the first pair of parentheses
    //     in the first regex match
    //   - the part of the string captured in the second pair of parentheses
    //     in the first regex match
    //   - etc. until the last pair of parentheses in the first match
    // - the part of the string after the first match, and before the 2nd match
    //   - capturing parentheses for the second match
    // - etc. for all regex matches
    // - part of the string after the last regex match
    // Tips: If you want the delimiters to be separate items in the array,
    //       put round brackets around the entire regex.
    //       If you need parentheses for grouping, but don't want their results
    //       in the array, use (?:subregex) non-capturing parentheses.
    //       If you want the delimiters to be included with the split items
    //       in the array, use lookahead or lookbehind to match a position
    //       in the string rather than characters.
    // E.g.: The regex "," separates a comma-delimited list, deleting the commas
    //       The regex "(,)" separates a comma-delimited list, inserting the
    //       commas as separate strings into the array of strings.
    //       The regex "(?<=,)" separates a comma-delimited list, leaving the
    //       commas at the end of each string in the array.
    // You can omit the last parameter with the regex options
    // if you don't want to specify any.
    textReplaceResults.Text = "N/A";
    try
    {
      printSplitArray(Regex.Split(textSubject.Text, textRegex.Text,
                                  getRegexOptions()));
    }
    catch (Exception ex)
    {
      // Most likely cause is a syntax error in the regular expression
      textResults.Text = "Regex.Split() threw an exception:\r\n" + ex.Message;
    }
  }

  private Regex regexObj;
  private Match matchObj;

  private void printMatch()
  {
    // Regex.Match constructs and returns a Match object
    // You can query this object to get all possible information about the match
    if (matchObj.Success) {
      textResults.Text = "Match offset: " + matchObj.Index.ToString() + "\r\n";
      textResults.Text += "Match length: " + matchObj.Length.ToString() +"\r\n";
      textResults.Text += "Matched text: " + matchObj.Value + "\r\n";
      if (matchObj.Groups.Count > 1) {
        // matchObj.Groups[0] holds the entire regex match also held by
        // matchObj itself.  The other Group objects hold the matches for
        // capturing parentheses in the regex
        for (int i = 1; i < matchObj.Groups.Count; i++) {
          Group g = matchObj.Groups[i];
          if (g.Success) {
```

```
          textResults.Text += "Group " + i.ToString() +
                              " offset: " + g.Index.ToString() + "\r\n";
          textResults.Text += "Group " + i.ToString() +
                              " length: " + g.Length.ToString() + "\r\n";
          textResults.Text += "Group " + i.ToString() +
                              " text: " + g.Value + "\r\n";
        } else {
          textResults.Text += "Group " + i.ToString() +
                              " did not participate in the overall match\r\n";
        }
      }
    } else {
      textResults.Text += "no backreferences/groups";
    }
  } else {
    textResults.Text = "no match";
  }
  textReplaceResults.Text = "N/A";
}

private void btnRegexObj_Click(object sender, System.EventArgs e)
{
  // Clean up, in case we cannot construct the new regex object
  regexObj = null;
  textReplaceResults.Text = "N/A";
  // If you want to do many searches using the same regular expression,
  // you should first construct a System.Text.RegularExpressions.Regex object
  // and then call its Match method (one of the overloaded forms that does
  // not take the regular expression as a parameter)
  // regexOptions may be omitted if all options are off
  try
  {
    regexObj = new Regex(textRegex.Text, getRegexOptions());
    textResults.Text = "Regex object constructed.  Click on one of the " +
      "buttons to the right of the Create Object button to use the object.";
  }
  catch (Exception ex)
  {
    // Most likely cause is a syntax error in the regular expression
    textResults.Text = "Regex constructor threw an exception:\r\n"
                       + ex.Message;
    return;
  }
}

private void btnFirstMatch_Click(object sender, System.EventArgs e)
{
  // Find the first match using regexObj constructed in btnRegexObj_Click()
  // and store all the details in matchObj
  // matchObj is used in btnNextMatch_click() to find subsequent matches
  if (regexObj == null) {
    textResults.Text = "First click on Create Object to create the regular " +
      "expression object.  Then click on First Match to find " +
      "the first match in the subject string.";
    textReplaceResults.Text = "N/A";
  } else {
    matchObj = regexObj.Match(textSubject.Text);
    printMatch();
  }
}

private void btnNextMatch_Click(object sender, System.EventArgs e)
{
  // Tell the regex engine to find another match after the previous match
  // Note that even if you change textRegex.Text or textSubject.Text between
  // clicking btnRegexObj, btnFirstMatch and btnNextMatch, the regex engine
  // will continue to search the same subject string passed in the
  // regexObj.Match call in btnFirstMatch_Click using the same regular
  // expression passed to the Regex() constructor in btnRegexObj_Click
  if (matchObj == null) {
    textResults.Text = "Use the First Match button to find the 1st match." +
```

```
                        "Then use this button to find following matches.";
      textReplaceResults.Text = "N/A";
    } else {
      matchObj = matchObj.NextMatch();
      printMatch();
    }
  }

  private void btnRegexObjReplace_Click(object sender, System.EventArgs e)
  {
    // If you want to do many search-and-replace operations using the same
    // regular expression, you should first construct a
    // System.Text.RegularExpressions.Regex object and then call its Replace()
    // method (one of the overloaded forms that does not take the regular
    // expression as a parameter).
    // This way also allows to to specify two additional parameters allowing
    // you to control how many replacements will be made.
    // The easy way used in btnReplace_click will always replace ALL matches.
    // See the comments with btnReplace_click for explanation of the special
    // $-placeholders you can use in the replacement text.
    // You can mix calls to regexObj.Match() and regexObj.Replace() as you like.
    // The results of the calls will not affect the other calls.
    if (regexObj == null) {
      textReplaceResults.Text = "Please use the Create Objects button to" +
        "construct the regex object.\r\n" +
        "Then use this button to do a search-and-replace using the subject" +
        "and replacement texts.";
    } else {
      // As used in this example, Replace() will replace ALL matches of the
      // regex in the subject with the replacement text.
      // If you want to limit the number of matches replaced, specify a third
      // parameter with the number of matches to be replaced.
      // If you specify 3, the first (left-to-right) 3 matches will be replaced.
      // You can also specify a fourth parameter with the character position
      // in the subject where the regex search should begin.
      // If the third parameter is negative, all matches after the starting
      // position will be replaced like when the third and fourth parameters
      // are omitted.
      textReplaceResults.Text = regexObj.Replace(textSubject.Text,
        textReplace.Text /*, ReplaceCount, ReplaceStart*/ );
    }
    textResults.Text = "N/A";
  }

  private void btnRegexObjSplit_Click(object sender, System.EventArgs e)
  {
    // If you want to split many strings using the same regular expression,
    // you should first construct a System.Text.RegularExpressions.Regex object
    // and then call its Split method (one of the overloaded forms that does
    // not take the regular expression as a parameter).
    // See btnSplit_Click for an explanation how Split() works.
    // If you first construct a Regex object, you can specify two additional
    // parameters to Split() after the subject string.
    // The optional second parameter indicates how many times Split() is
    // allowed to split the string.  A negative number causes the string to be
    // split at all regex matches.  If the number is smaller than the number
    // of possible matches, then the last string in the returned array
    // will contain the unsplit remainder of the string.
    // The optional third parameter indicates the character position in the
    // string where Split() can start to look for regex matches.
    // If you specify the third parameter, then the first string in the returned
    // array will contain the unsplit start of the string as well as
    // the part of the string between the starting position and the first match.
    // You can mix calls to regexObj.Match() and regexObj.Split() as you like.
    // The results of the calls will not affect the other calls.
    textReplaceResults.Text = "N/A";
    if (regexObj == null) {
      textResults.Text = "Please use the Create Objects button to construct" +
        "the regular expression object.\r\n" +
        "Then use this button to split the subject into an array of strings.";
    } else {
```

```
        printSplitArray(regexObj.Split(textSubject.Text)
                        /*, SplitCount, SplitStart*/ );
      }
    }

  }
}
```

10. The PCRE Open Source Regex Library

PCREis short for Perl Compatible Regular Expressions. It is the name of an open source library written in C by Phillip Hazel. The library is compatible with a great number of C compilers and operating systems. Many people have derived libraries from PCRE to make it compatible with other programming languages. E.g. there are several Delphi components that are simply wrappers around the PCRE library compiled into a Win32 DLL. The library is also included with many Linux distributions as a shared .so library and a .h header file.

PCRE implements almost the entire Perl 5 regular expression syntax. Its use is very straightforward. Before you can use a regular expression, it needs to be converted into a binary format for improved efficiency. To do this, simply call pcre_compile() passing your regular expression as a null-terminated string. The function will return a pointer to the binary format. You cannot do anything with the result except pass it to the other pcre functions.

To use the regular expression, call pcre_exec() passing the pointer returned by pcre_compile(), the character array you want to search through, and the number of characters in the array (which need not be null-terminated). You also need to pass a pointer to an array of integers where pcre_exec() will store the results, as well as the length of the array expressed in integers. The length of the array should equal the number of capturing groups you want to support, plus one (for the entire regex match), multiplied by three (!). The function will return 0 if no match could be found. Otherwise, it will return the number of capturing groups filled plus one. The first two integers in the array with results contain the start of the regex match (counting bytes from the start of the array) and the number of bytes in the regex match, respectively. The following pairs of integers contain the start and length of the backreferences. So array[n*2] is the start of capturing group n, and array[n*2+1] is the length of capturing group n, with capturing group 0 being the entire regex match.

When you are done with a regular expression, all pcre_dispose() with the pointer returned by pcre_compile() to prevent memory leaks.

The PCRE library only supports regex matching, a job it does rather well. It provides no support for search-and-replace, splitting of strings, etc. This is not a major issue, as you can easily do that in your own code.

You can find more information about PCRE on http://www.pcre.org/.

11. Perl's Rich Support for Regular Expressions

Perl was originally designed by Larry Wall as a flexible text-processing language. Over the years, it has grown into a full-fledged programming language, keeping a strong focus on text processing. When the world wide web became popular, Perl became the de facto standard for creating CGI scripts. A CGI script is a small piece of software that generates a dynamic web page, based on a database and/or input from the person visiting the web site. Since CGI script basically is a text-processing script, Perl was and still is a natural choice.

Because of Perl's focus on managing and mangling text, regular expression text patterns are an integral part of the Perl language. This in contrast with most other languages, where regular expressions are available as add-on libraries. In Perl, you can use the `m//` operator to test if a regex can match a string, e.g.:

```
if ($string =~ m/regex/) {
  print 'match';
} else {
  print 'no match';
}
```

Performing a regex search-and-replace is just as easy:

```
$string =~ s/regex/replacement/g;
```

I added a “g” after the last forward slash. The “g” stands for “global”, which tells Perl to replace all matches, and not just the first one. Options are typically indicated including the slash, like "`/g`", even though you do not add an extra slash, and even though you could use any non-word character instead of slashes. If your regex contains slashes, use another character, like `s!regex!replacement!g`.

You can add an “i” to make the regex match case insensitive. You can add an “s” to make the dot match newlines. You can add an “m” to make the dollar and caret match at newlines embedded in the string, as well as at the start and end of the string.

Together you would get something like `m/regex/sim;`

Regex-Related Special Variables

Perl has a host of special variables that get filled after every `m//` or `s///` regex match. `$1`, `$2`, `$3`, etc. hold the backreferences. `$+` holds the last (highest-numbered) backreference. `$&` (dollar ampersand) holds the entire regex match.

`@-` is an array of match-start indices into the string. `$-[0]` holds the start of the entire regex match, `$-[1]` the start of the first backreference, etc. Likewise, `@+` holds match-end indices (ends, not lengths).

`$'` (dollar followed by an apostrophe or single quote) holds the part of the string after (to the right of) the regex match. `$`` (dollar backtick) holds the part of the string before (to the left of) the regex match. Using these variables is not recommended in scripts when performance matters, as it causes Perl to slow down *all* regex matches in your entire script.

All these variables are read-only, and persist until the next regex match is attempted. They are dynamically scoped, as if they had an implicit 'local' at the start of the enclosing scope. Thus if you do a regex match, and

call a sub that does a regex match, when that sub returns, your variables are still set as they were for the first match.

Finding All Matches In a String

The "/g" modifier can be used to process all regex matches in a string. The first `m/regex/g` will find the first match, the second `m/regex/g` the second match, etc. The location in the string where the next match attempt will begin is automatically remembered by Perl, separately for each string. Here is an example:

```
while ($string =~ m/regex/g) {
  print "Found '$&'.  Next attempt at character " . pos($string)+1 . "\n";
}
```

The `pos()` function retrieves the position where the next attempt begins. The first character in the string has position zero. You can modify this position by using the function as the left side of an assignment, like in `pos($string) = 123;`.

More Information

The above describes how you can use regular expressions with Perl, and is probably all you need to know. But if you want to get in-depth information of all the regex-related tricks Perl can perform, I recommend you pick up a copy of the second edition of Jeffrey Friedl's Mastering Regular Expressions. It has an interesting 80-page chapter on regex-related Perl oddities. General Perl books usually only contain the general stuff that is better explained in the tutorial in this book .

12. PHP Provides Two Sets of Regular Expression Functions

PHP is an open source language for producing dynamic web pages, similar to ASP. PHP has two sets of functions that allow you to work with regular expressions.

The first set of regex functions are those that start with `ereg`. They implement a basic set of regular expression functionality, much like the traditional UNIX egrep command. The advantage of the `ereg` functions is that they are supported by all but the oldest versions of PHP. However, many of the more modern regex features such as lazy quantifiers and lookaround are not supported by the `ereg` functions.

The second set of regex functions start with `preg`. These functions are only available if your version of PHP was compiled with support for the PCRE library, and the PCRE library is installed on your web server. Just like the PCRE library, the `preg` functions support the complete regular expression syntax described by the regular expression tutorial in this book .

If you are developing PHP scripts that will be used by others on their own servers, I recommend that you restrict yourself to the `ereg` functions, for maximum compatibility. But if you know the servers your script will be used on support the `preg` functions, then by all means use them.

The ereg Function Set

The `ereg` functions require you to specify the regular expression as a string, as you would expect. `ereg('regex', "subject")` checks if «`regex`» matches "`subject`". You should use single quotes when passing a regular expression as a literal string. Several special characters like the dollar and backslash are also special characters in double-quoted PHP strings, but not in single-quoted PHP strings.

`bool` **`ereg`** `(string pattern, string subject [, array groups])` returns TRUE if the regular expression pattern matches the subject string or part of the subject string. If you specify the third parameter, `ereg` will store the substring matched by the part of the regular expression between the first pair of round brackets in $groups[1]. $groups[2] will contain the second pair, and so on. Note that grouping-only round brackets are not supported by ereg. `ereg` is case sensitive. `eregi` is the case insensitive equivalent.

`string` **`ereg_replace`** `(string pattern, string replacement, string subject)` replaces all matches of the regex patten in the subject string with the replacement string. You can use backreferences in the replacement string. `\\0` is the entire regex match, `\\1` is the first backreference, `\\2` the second, etc. The highest possible backreference is `\\9`. `ereg_replace` is case sensitive. `eregi_replace` is the case insensitive equivalent.

`array` **`split`** `(string pattern, string subject [, int limit])` splits the subject string into an array of strings using the regular expression pattern. The array will contain the substrings between the regular expression matches. The text actually matched is discarded. If you specify a limit, the resulting array will contain at most that many substrings. The subject string will be split at most limit-1 times, and the last item in the array will contain the unsplit remainder of the subject string. `split` is case sensitive. `spliti` is the case insensitive equivalent.

See the PHP manual for more information on the ereg function set

The preg Function Set

All of the `preg` functions require you to specify the regular expression as a string using Perl syntax. In Perl, `/regex/` defines a regular expression. In PHP, this becomes `preg_match('/regex/', $subject)`. Forward slashes in the regular expression have to be escaped with a backslash. So «`http://www.jgsoft.com/`» becomes `'/http:\/\/www.jgsoft.com\//'`.

To specify regex matching options such as case insensitivity are specified in the same way as in Perl. `'/regex/i'` applies the regex case insensitively. `'/regex/s'` makes the dot match all characters. `'/regex/m'` makes the start and end of line anchors match at embedded newlines in the subject string. You can specify multiple letters to turn on several options. `'/regex/mis'` turns on all three options. See the PHP manual for the complete list of options. Mostly, you will only use these three.

Like the `ereg` function, `bool` **`preg_match`** `(string pattern, string subject [, array groups])` returns TRUE if the regular expression pattern matches the subject string or part of the subject string. If you specify the third parameter, `preg` will store the substring matched by the part of the regular expression between the first pair of capturing parentheses in $groups[1]. $groups[2] will contain the second pair, and so on. If the regex pattern uses named capture, you can access the groups by name with `$groups['name']`.

`int` **`preg_match_all`** `(string pattern, string subject, array matches, int flags)` fills the array "matches" with all the matches of the regular expression pattern in the subject string. If you specify PREG_SET_ORDER as the flag, then `$matches[0]` is an array containing the match and backreferences of the first match, just like the `$groups` array filled by `preg_match`. `$matches[1]` holds the results for the second match, and so on. If you specify PREG_PATTERN_ORDER, then `$matches[0]` is an array with full subsequent regex matches, `$matches[1]` an array with the first backreference of all matches, `$matches[2]` an array with the second backreference of each match, etc.

`array` **`preg_grep`** `(string pattern, array subjects)` returns an array that contains all the strings in the array "subjects" that can be matched by the regular expression pattern.

Like `ereg_replace`, `mixed` **`preg_replace`** `(mixed pattern, mixed replacement, mixed subject [, int limit])` returns a string with all matches of the regex pattern in the subject string replaced with the replacement string. At most `limit` replacements are made. One key difference is that all parameters, except limit, can be arrays instead of strings. In that case, `preg_replace` does its job multiple times, iterating over the elements in the arrays simultaneously. You can also use strings for some parameters, and arrays for others. Then the function will iterate over the arrays, and use the same strings for each iteration. Using an array of the pattern and replacement, allows you to perform a sequence of search and replace operations on a single subject string. Using an array for the subject string, allows you to perform the same search and replace operation on many subject strings.

`array` **`preg_split`** `(string pattern, string subject [, int limit])` works just like `split`, except that it uses the Perl syntax for the regex pattern.

See the PHP manual for more information on the preg function set

13. Python's re Module

Python is a high level open source scripting language. Python's built-in “re” module provides excellent support for regular expressions, with a modern and complete regex flavor,. The only feature currently missing from Python's regex syntax are atomic grouping and possessive quantifiers.

The first thing to do is to import the regexp module into your script with `import re`.

Regex Search and Match

Call `re.search(regex, subject)` to apply a regex pattern to a subject string. The function returns `None` if the matching attempt fails, and a `Match` object otherwise. Since `None` evaluates to `False`, you can easily use `re.search()` in an `if` statement. The `Match` object stores details about the part of the string matched by the regular expression pattern.

You can set regex matching modes by specifying a special constant as a third parameter to `re.search()`. `re.I` or `re.IGNORECASE` applies the pattern case insensitively. `re.S` or `re.DOTALL` makes the dot match newlines. `re.M` or `re.MULTILINE` makes the caret and dollar match after and before line breaks in the subject string. There is no difference between the single-letter and descriptive options, except for the number of characters you have to type in. To specify more than one option, “or” them together with the `|` operator: `re.search("^a", “abc”, re.I | re.M)`.

By default, Python's regex engine only considers the letters A through Z, the digits 0 through 9, and the underscore as “word characters”. Specify the flag `re.L` or `re.LOCALE` to make «`\w`» match all characters that are considered letters given the current locale settings. Alternatively, you can specify `re.U` or `re.UNICODE` to treat all letters from all scripts as word characters. The setting also affects word boundaries.

Do not confuse `re.search()` with `re.match()`. Both functions do exactly the same, with the important distinction that `re.search()` will attempt the pattern throughout the string, until it finds a match. `re.match()` on the other hand, only attempts the pattern at the very start of the string. Basically, `re.match(“regex”, subject)` is the same as `re.search("\Aregex", subject)`. Note that `re.match()` does *not* require the regex to match the entire string. `re.match(“a”, “ab”)` will succeed.

To get all matches from a string, call `re.findall(regex, subject)`. This will return an array of all non-overlapping regex matches in the string. “Non-overlapping” means that the string is searched through from left to right, and the next match attempt starts beyond the previous match. If the regex contains one or more capturing groups, `re.findall()` returns an array of tuples, with each tuple containing text matched by all the capturing groups. The overall regex match is *not* included in the tuple, unless you place the entire regex inside a capturing group.

More efficient than `re.findall()` is `re.finditer(regex, subject)`. It returns an iterator that enables you to loop over the regex matches in the subject string: `for m in re.finditer(regex, subject)`. The for-loop variable `m` is a `Match` object with the details of the current match.

Unlike `re.search()` and `re.match()`, `re.findall()` and `re.finditer()` do not support an optional third parameter with regex matching flags. Instead, you can use global mode modifiers at the start of the regex. E.g. "(?i)regex" matches «`regex`» case insensitively.

Strings, Backslashes and Regular Expressions

The backslash is a metacharacter in regular expressions, and is used to escape other metacharacters. The regex «\\» matches a single backslash. «\d» is a single token matching a digit.

Python strings also use the backslash to escape characters. The above regexes are written as Python strings as "\\\\“ and ”\\w". Confusing indeed.

Fortunately, Python also has “raw strings” which do not apply special treatment to backslashes. As raw strings, the above regexes become r"\\“ and r”\w". The only limitation of using raw strings is that the delimiter you're using for the string must not appear in the regular expression, as raw strings do not offer a means to escape it.

You can use \n and \t in raw strings. Though raw strings do not support these escapes, the regular expression engine does. The end result is the same.

Search and Replace

`re.sub(regex, replacement, subject)` performs a search-and-replace across subject, replacing all matches of regex in subject with replacement. The result is returned by the `sub()` function. The subject string you pass is not modified.

If the regex has capturing groups, you can use the text matched by the part of the regex inside the capturing group. To substitute the text from the third group, insert `\3` into the replacement string. If you want to use the text of the third group followed by a literal zero as the replacement, use the string `r"\g<3>3"`. `\33` is interpreted as the 33rd group, and is substituted with nothing if there are fewer groups. If you used named capturing groups, you can use them in the replacement text with `r"\g<name>"`.

The `re.sub()` function applies the same backslash logic to the replacement text as is applied to the regular expression. Therefore, you should use raw strings for the replacement text, as I did in the examples above. The `re.sub()` function will also interpret \n and \t in raw strings. If you want “`c:\temp`” as the replacement text, either use `r"c:\\temp"` or `"c:\\\\temp"`. The 3rd backreferenence is `r"\3"` or `"\\3"`.

Splitting Strings

`re.split(regex, subject)` returns an array of strings. The array contains the parts of subject between all the regex matches in the subject. Adjacent regex matches will cause empty strings to appear in the array. The regex matches themselves are not included in the array. If the regex contains capturing groups, then the text matched by the capturing groups is included in the array. The capturing groups are inserted between the substrings that appeared to the left and right of the regex match. If you don't want the capturing groups in the array, convert them into non-capturing groups. The `re.split()` function does not offer an option to suppress capturing groups.

You can specify an optional third parameter to limit the number of times the subject string is split. Note that this limit controls the number of splits, not the number of strings that will end up in the array. The unsplit remainder of the subject is added as the final string to the array. If there are no capturing groups, the array will contain `limit+1` items.

Match Details

`re.search()` and `re.match()` return a Match object, while `re.finditer()` generates an iterator to iterate over a Match object. This object holds lots of useful information about the regex match. I will use `m` to signify a Match object in the discussion below.

`m.group()` returns the part of the string matched by the entire regular expression. `m.start()` returns the offset in the string of the start of the match. `m.end()` returns the offset of the character beyond the match. `m.span()` returns a 2-tuple of `m.start()` and `m.end()`. You can use the `m.start()` and `m.end()` to slice the subject string: `subject[m.start():m.stop()]`.

If you want the results of a capturing group rather than the overall regex match, specify the name or number of the group as a parameter. `m.group(3)` returns the text matched by the third capturing group. `m.group('groupname')` returns the text matched by a named group 'groupname'. If the group did not participate in the overall match, `m.group()` returns an empty string, while `m.start()` and `m.end()` return -1.

If you want to do a regular expression based search-and-replace without using `re.sub()`, call `m.expand(replacement)` to compute the replacement text. The function returns the replacement string with backreferences etc. substituted.

Regular Expression Objects

If you want to use the same regular expression more than once, you should compile it into a regular expression object. Regular expression objects are more efficient, and make your code more readable. To create one, just call `re.compile(regex)` or `re.compile(regex, flags)`. The flags are the matching options described above for the `re.search()` and `re.match()` functions.

The regular expression object returned by `re.compile()` provides all the functions that the re module also provides directly: `search()`, `match()`, `findall()`, `finditer()`, `sub()` and `split()`. The difference is that they use the pattern stored in the regex object, and do not take the regex as the first parameter. `re.compile(regex).search(subject)` is equivalent to `re.search(regex, subject)`.

14. Using Regular Expressions with Ruby

Ruby supports regular expressions as a language feature. In Ruby, a regular expression is written in the form of `/pattern/modifiers` where “pattern” is the regular expression itself, and “modifiers” are a series of characters indicating various options. The “modifiers” part is optional. This syntax is borrowed from Perl. Ruby supports the following modifiers:

- `/i` makes the regex match case insensitive.
- `/m` makes the dot match newlines. Ruby indeed uses `/m`, whereas Perl and many other programming languages use `/s` for "dot matches newlines".
- `/o` causes any `#{...}` substitutions in a particular regex literal to be performed just once, the first time it is evaluated. Otherwise, the substitutions will be performed every time the literal generates a Regexp object.

You can combine multiple modifiers by stringing them together as in `/regex/is`.

In Ruby, the caret and dollar always match before and after newlines. Ruby does not have a modifier to change this. Use «`\A`» and «`\Z`» to match at the start or the end of the string.

Since forward slashes delimit the regular expression, any forward slashes that appear in the regex need to be escaped. E.g. the regex «`1/2`» is written as `/1\/2/` in Ruby.

How To Use The Regexp Object

`/regex/` creates a new object of the class Regexp. You can assign it to a variable to repeatedly use the same regular expression, or use the literal regex directly. To test if a particular regex matches (part of) a string, you can either use the `=~` operator, call the regexp object's `match()` method, e.g.: `print “success” if subject =~ /regex/` or `print “success” if /regex/.match(subject)`.

The `=~` operator returns the character position in the string of the start of the match (which evaluates to true in a boolean test), or nil if no match was found (which evaluates to false). The `match()` method returns a MatchData object (which also evaluates to true), or nil if no matches was found. In a string context, the MatchData object evaluates to the text that was matched. So `print(/\w+/.match(“test”))` prints “test”, while `print(/\w+/ =~ “test”)` prints “0”. The first character in the string has index zero. Switching the order of the `=~` operator's operands makes no difference.

Search And Replace

Use the `sub()` and `gsub()` methods of the String class to search-and-replace the first regex match, or all regex matches, respectively, in the string. Specify the regular expression you want to search for as the first parameter, and the replacement string as the second parameter, e.g.: `result = subject.gsub(/before/, “after”)`.

To re-insert the regex match, use `\0` in the replacement string. You can use the contents of capturing groups in the replacement string with backreferences `\1`, `\2`, `\3`, etc. Note that numbers escaped with a backslash are treated as octal escapes in double-quoted strings. Octal escapes are processed at the language level, before the sub() function sees the parameter. To prevent this, you need to escape the backslashes in double-quoted

strings. So to use the first backreference as the replacement string, either pass `'\1'` or `"\\1"`. `'\\1'` also works.

Splitting Strings and Collecting Matches

To collect all regex matches in a string into an array, pass the regexp object to the string's `scan()` method, e.g.: `myarray = mystring.scan(/regex/)`. Sometimes, it is easier to create a regex to match the delimiters rather than the text you are interested in. In that case, use the `split()` method instead, e.g.: `myarray = mystring.scan(/delimiter/)`. The `split()` method discards all regex matches, returning the text between the matches. The `scan()` method does the opposite.

If your regular expression contains capturing groups, `scan()` returns an array of arrays. Each element in the overall array will contain an array consisting of the overall regex match, plus the text matched by all capturing groups.

15. VBScript's Regular Expression Support

VBScript has built-in support for regular expressions. If you use VBScript to validate user input on a web page at the client side, using VBScript's regular expression support will greatly reduce the amount of code you need to write.

Microsoft made some significant enhancements to VBScript's regular expression support in version 5.5 of Internet Explorer. Version 5.5 implements quite a few essential regex features that were missing in previous versions of VBScript. Internet Explorer 6.0 does not expand the regular expression functionality. Whenever this book mentions VBScript, the statements refer to VBScript's version 5.5 regular expression support.

You can use regular expressions in VBScript by creating one or more instances of the `RegExp` object. This object allows you to find regular expression matches in strings, and replace regex matches in strings with other strings. The functionality offered by VBScript's RegExp object is pretty much bare bones. However, it's more than enough for simple input validation and output formatting tasks typically done in VBScript.

VBScript's regular expression flavor is the same Perl-style flavor used by all other programming languages discussed in this book. However, even version 5.5 lacks quite a number of advanced features:

- No \A or \Z anchors to match the start or end of the string. Use a caret or dollar instead.
- Lookbehind is not supported at all. Lookahead is fully supported.
- No atomic grouping or possessive quantifiers
- No Unicode support
- No named capturing groups. Use numbered capturing groups instead.
- No regular expression comments. Describe your regular expression with VBScript apostrophe comments instead, outside the regular expression string.

Version 1.0 of the RegExp object even lacks basic features like lazy quantifiers. This is the main reason this book does not discuss VBScript RegExp 1.0. All versions of Internet Explorer prior to 5.5 include version 1.0 of the RegExp object. There are no other versions than 1.0 and 5.5.

How to Use the VBScript RegExp Object

The advantage of the RegExp object's bare-bones nature is that it's very easy to use. Create one, put in a regex, and let it match or replace. Only four properties and three methods are available.

After creating the object, assign the regular expression you want to search for to the **`Pattern`** property. If you want to use a literal regular expression rather than a user-supplied one, simply put the regular expression in a double-quoted string. By default, the regular expression is case sensitive. Set the **`IgnoreCase`** property to True to make it case insensitive. The caret and dollar only match at the very start and very end of the subject string by default. If your subject string consists of multiple lines separated by line breaks, you can make the caret and dollar match at the start and the end of those lines by setting the **`Multiline`** property to True. VBScript does not have an option to make the dot match line break characters. Finally, if you want the RegExp object to return or replace all matches instead of just the first one, set the **`Global`** property to True.

```
Set myRegExp = New RegExp       'Prepare a regular expression object
myRegExp.IgnoreCase = True
myRegExp.Global = True
myRegExp.Pattern = "regex"
```

After setting the RegExp object's properties, you can invoke one of the three methods to perform one of three basic tasks. The **`Test`** method takes one parameter: a string to test the regular expression on. `Test` returns True or False, indicating if the regular expression matches (part of) the string. When validating user input, you'll typically want to check if the *entire* string matches the regular expression. To do so, put a caret at the start of the regex, and a dollar at the end, to anchor the regex at the start and end of the subject string.

The **`Execute`** method also takes one string parameter. Instead of returning True or False, it returns a `MatchCollection` object. If the regex could not match the subject string at all, `MatchCollection.Count` will be zero. If the `RegExp.Global` property is False (the default), `MatchCollection` will contain only the first match. If `RegExp.Global` is true, `Matches>` will contain all matches.

The **`Replace`** method takes two string parameters. The first parameter is the subject string, while the second parameter is the replacement text. If the `RegExp.Global` property is False (the default), `Replace` will return the subject string with the first regex match (if any) substituted with the replacement text. If `RegExp.Global` property is true, `Matches>` will contain all matches. If `RegExp.Global` is true, `Replace>` will return the subject string with all matches replaced.

You can specify an empty string as the replacement text. This will cause the `Replace` method to return the subject string will all regex matches deleted from it. To re-insert the regex match as part of the replacement, include "`$&`" in the replacement text. E.g. to enclose each regex match in the string between square brackets, specify "`[$&]`" as the replacement text. If the regexp contains capturing parentheses, you can use backreferences in the replacement text. `$1` in the replacement text inserts the text matched by the first capturing group, `$2` the second, etc. up to `$9`. To include a literal dollar sign in the replacements, put two consecutive dollar signs in the string you pass to the `Replace` method.

Getting Information about Individual Matches

The **`MatchCollection`** object returned by the `RegExp.Execute` method is a collection of `Match` objects. It has only two read-only properties. The **`Count`** property indicates how many matches the collection holds. The **`Item`** property takes an index parameter (ranging from zero to `Count-1`), and returns a `Match` object. The `Item` property is the default member, so you can write `MatchCollection(7)` as a shorthand to `MatchCollection.Item(7)`.

The easiest way to process all matches in the collection is to use a For Each construct, e.g.:

```
' Pop up a message box for each match
Set myMatches = myRegExp.Execute(subjectString)
For Each myMatch in myMatches
  msgbox myMatch.Value, 0, "Found Match"
Next
```

The **`Match`** object has four read-only properties. The **`FirstIndex`** property indicates the number of characters in the string to the left of the match. If the match was found at the very start of the string, `FirstIndex` will be zero. If the match starts at the second character in the string, `FirstIndex` will be one, etc. Note that this is different from the VBScript `Mid` function, which extracts the first character of the string if you set the `start` parameter to one. The **`Length`** property of the `Match` object indicates the number of characters in the match. The **`Value`** property returns the text that was matched.

The **`SubMatches`** property of the `Match` object is a collection of strings. It will only hold values if your regular expression has capturing groups. The collection will hold one string for each capturing group. The

`Count` property indicates the number of string in the collection. The **`Item`** property takes an index parameter, and returns the text matched by the capturing group. The `Item` property is the default member, so you can write `SubMatches(7)` as a shorthand to `SubMatches.Item(7)`. Unfortunately, VBScript does not offer a way to retrieve the match position and length of capturing groups.

Also unfortunately is that the SubMatches property does *not* hold the complete regex match as `SubMatches(0)`. Instead, SubMatches(0) holds the text matched by the first capturing group, while SubMatches(SubMatches.Count-1) holds the text matched by the last capturing group. This is different from most other programming languages. E.g. in VB.NET, `Match.Groups(0)` returns the whole regex match, and `Match.Groups(1)` returns the first capturing group's match. Note that this is also different from the backreferences you can use in the replacement text passed to the `RegExp.Replace` method. In the replacement text, $1 inserts the text matched by the first capturing group, just like most other regex flavors do. $0 is not substituted with anything but inserted literally.

16. VBScript RegExp Example: Regular Expression Tester

```
<SCRIPT LANGUAGE="VBScript"><!--
Sub btnTest_OnClick
  Set re = New RegExp
  re.Pattern = document.demoMatch.regex.value
  If re.Test(document.demoMatch.subject.value) Then
    msgbox "Successful match", 0, "VBScript Regular Expression Tester"
  Else
    msgbox "No match", 0, "VBScript Regular Expression Tester"
  End If
End Sub

Sub btnMatch_OnClick
  Set re = New RegExp
  re.Pattern = document.demoMatch.regex.value
  Set matches = re.Execute(document.demoMatch.subject.value)
  If matches.Count > 0 Then
    Set match = matches(0)
    msg = "Found match """ & match.Value & _
          """ at position " & match.FirstIndex & vbCRLF
    If match.SubMatches.Count > 0 Then
      For I = 0 To match.SubMatches.Count-1
        msg = msg & "Group #" & I+1 & " matched """ & _
              match.SubMatches(I) & """" & vbCRLF
      Next
    End If
    msgbox msg, 0, "VBScript Regular Expression Tester"
  Else
    msgbox "No match", 0, "VBScript Regular Expression Tester"
  End If
End Sub

Sub btnMatchGlobal_OnClick
  Set re = New RegExp
  re.Pattern = document.demoMatch.regex.value
  re.Global = True
  Set matches = re.Execute(document.demoMatch.subject.value)
  If matches.Count > 0 Then
    msg = "Found " & matches.Count & " matches:" & vbCRLF
    For Each match In Matches
      msg = msg & "Found match """ & match.Value & _
            """ at position " & match.FirstIndex & vbCRLF
    Next
    msgbox msg, 0, "VBScript Regular Expression Tester"
  Else
    msgbox "No match", 0, "VBScript Regular Expression Tester"
  End If
End Sub

Sub btnReplace_OnClick
  Set re = New RegExp
  re.Pattern = document.demoMatch.regex.value
  re.Global = True
  document.demoMatch.result.value = _
    re.Replace(document.demoMatch.subject.value, _
    document.demoMatch.replacement.value)
End Sub

' -->
</SCRIPT>
```

```
<FORM ID="demoMatch" NAME="demoMatch">
<P>Regexp: <INPUT TYPE=TEXT NAME="regex" VALUE="\bt[a-z]+\b" SIZE=50></P>
<P>Subject string: <INPUT TYPE=TEXT NAME="subject"
   VALUE="This is a test of the VBScript RegExp object" SIZE=50></P>
<P><INPUT TYPE=BUTTON NAME="btnTest" VALUE="Test Match">
<INPUT TYPE=BUTTON NAME="btnMatch" VALUE="Show Match">
<INPUT TYPE=BUTTON NAME="btnMatchGlobal" VALUE="Show All Matches"></P>

<P>Replacement text: <INPUT TYPE=TEXT NAME="replacement"
   VALUE="replaced" SIZE=50></P>
<P>Result: <INPUT TYPE=TEXT NAME="result"
   VALUE="click the button to see the result" SIZE=50></P>
<P><INPUT TYPE=BUTTON NAME="btnReplace" VALUE="Replace"></P>
</FORM>
```

17. How to Use Regular Expressions in Visual Basic

Unlike Visual Basic.NET, which has access to the excellent regular expression support of the .NET framework, good old Visual Basic 6 does not ship with any regular expression support. However, VB6 does make it very easy to use functionality provided by ActiveX and COM libraries.

One such library is Microsoft's VBScript scripting library, which as decent regular expression capabilities starting with version 5.5. This library is part of Internet Explorer 5.5 and later. It is available on all computers running Windows XP (which ships with IE 6.0), and previous versions of Windows if the user upgraded to IE 5.5 or later. That includes almost every PC that is used to connect to the internet.

To use this library in your Visual Basic application, select Project|References in the VB IDE's menu. Scroll down the list to find the item "Microsoft VBScript Regular Expressions 5.5". It's immediately below the "Microsoft VBScript Regular Expressions 1.0" item. Make sure to tick the 5.5 version, *not* the 1.0 version. The 1.0 version is only provided for backward compatibility. It's capabilities are less than satisfactory.

After adding the reference, you can see which classes and class members the library provides. Select View|Object Browser in the menu. In the Object Browser, select the “VBScript_RegExp_55” library in the drop-down list in the upper left corner. For a detailed description, see the VBScript regular expression reference in this book.

The only difference between VB6 and VBScript is that you'll need to use a `Dim` statement to declare the objects prior to creating them. Here's a complete code snippet. It's the two code snippets on the VBScript page put together, with three `Dim` statements added.

```
'Prepare a regular expression object
Dim myRegExp As RegExp
Dim myMatches As MatchCollection
Dim myMatch As Match
Set myRegExp = New RegExp
myRegExp.IgnoreCase = True
myRegExp.Global = True
myRegExp.Pattern = "regex"
Set myMatches = myRegExp.Execute(subjectString)
For Each myMatch in myMatches
  MsgBox(myMatch.Value)
Next
```

Part 5

Regular Expression Reference

1. Basic Syntax Reference

Characters

Character: Any character except `[\^$.|?*+(`
Description: All characters except the listed special characters match a single instance of themselves.
Example: «a» matches „a”

Character: \ (backslash) followed by any of `[\^$.|?*+(`
Description: A backslash escapes special characters to suppress their special meaning.
Example: «\+» matches „+”

Character: `\xFF` where FF are 2 hexadecimal digits
Description: Matches the character with the specified ASCII/ANSI value, which depends on the code page used. Can be used in character classes.
Example: «`\xA9`» matches „©” when using the Latin-1 code page.

Character: `\n`, `\r` and `\t`
Description: Match an LF character, CR character and a tab character respectively. Can be used in character classes.
Example: «`\r\n`» matches a DOS/Windows CRLF line break.

Character Classes or Character Sets [abc]

Character: `[` (opening square bracket)
Description: Starts a character class. A character class matches a single character out of all the possibilities offered by the character class. Inside a character class, different rules apply. The rules in this section are only valid inside character classes. The rules outside this section are not valid in character classes, except `\n`, `\r`, `\t` and `\xFF`
Character: Any character except `^-]\` add that character to the possible matches for the character class.
Description: All characters except the listed special characters.
Example: «`[abc]`» matches „a”, „b” or „c”

Character: \ (backslash) followed by any of `^-]\`
Description: A backslash escapes special characters to suppress their special meaning.
Example: «`[\^\]]`» matches „^” or „]”

Character: `-` (hyphen) except immediately after the opening `[`
Description: Specifies a range of characters. (Specifies a hyphen if placed immediately after the opening `[`)
Example: «`[a-zA-Z0-9]`» matches any letter or digit

Character: `^` (caret) immediately after the opening `[`
Description: Negates the character class, causing it to match a single character *not* listed in the character class. (Specifies a caret if placed anywhere except after the opening `[`)
Example: «`[^a-d]`» matches „x” (any character except a, b, c or d)

Character: `\d`, `\w` and `\s`
Description: Shorthand character classes matching digits 0-9, word characters (letters and digits) and whitespace respectively. Can be used inside and outside character classes
Example: «`[\d\s]`» matches a character that is a digit or whitespace

Character: `\D`, `\W` and `\S`
Description: Negated versions of the above. Should be used only outside character classes. (Can be used inside, but that is confusing).)
Example: «`\D`» matches a character that is not a digit

Dot

Character: `.` (dot)
Description: Matches any single character except line break characters `\r` and `\n`. Most regex flavors have an option to make the dot match line break characters too.
Example: «`.`» matches „`x`” or (almost) any other character

Anchors

Character: `^` (caret)
Description: Matches at the start of the string the regex pattern is applied to. Matches a position rather than a character. Most regex flavors have an option to make the caret match after line breaks (i.e. at the start of a line in a file) as well.
Example: «`^.`» matches „`a`” in “`abc\ndef`”. Also matches „`d`” in "multi-line" mode.

Character: `$` (dollar)
Description: Matches at the end of the string the regex pattern is applied to. Matches a position rather than a character. Most regex flavors have an option to make the dollar match before line breaks (i.e. at the end of a line in a file) as well. Also matches before the very last line break if the string ends with a line break.
Example: «`.$`» matches „`f`” in “`abc\ndef`”. Also matches „`c`” in "multi-line" mode.

Character: `\A`
Description: Matches at the start of the string the regex pattern is applied to. Matches a position rather than a character. Never matches after line breaks.
Example: «`\A.`» matches „`a`” in “`abc`”

Character: `\Z`
Description: Matches at the end of the string the regex pattern is applied to. Matches a position rather than a character. Never matches before line breaks, except for the very last line break if the string ends with a line break.
Example: «`.\Z`» matches „`f`” in “`abc\ndef`”

Character: `\z`
Description: Matches at the end of the string the regex pattern is applied to. Matches a position rather than a character. Never matches before line breaks.
Example: «`.\z`» matches „`f`” in “`abc\ndef`”

Word Boundaries

Character: `\b`
Description: Matches at the position between a word character (anything matched by «`\w`») and a non-word character (anything matched by «`[^\w]`» or «`\W`») as well as at the start and/or end of the string if the first and/or last characters in the string are word characters.
Example: «`.\b`» matches „`c`” in “`abc`”

Character: `\B`
Description: Matches at the position between two word characters (i.e the position between «`\w\w`») as well as at the position between two non-word characters (i.e. «`\W\W`»).
Example: «`\B.\B`» matches „`b`” in “`abc`”

Alternation

Character: `|` (pipe)
Description: Causes the regex engine to match either the part on the left side, or the part on the right side. Can be strung together into a series of options.
Example: «`abc|def|xyz`» matches „`abc`”, „`def`” or „`xyz`”

Character: `|` (pipe)
Description: The pipe has the lowest precedence of all operators. Use grouping to alternate only part of the regular expression.
Example: «`abc(def|xyz)`» matches „`abcdef`” or „`abcxyz`”

Quantifiers

Character: `?` (question mark)
Description: Makes the preceding item optional. Greedy, so the optional item is included in the match if possible.
Example: «`abc?`» matches „`ab`” or „`abc`”

Character: `??`
Description: Makes the preceding item optional. Lazy, so the optional item is excluded in the match if possible. This construct is often excluded from documentation because of its limited use.
Example: «`abc??`» matches „`ab`” or „`abc`”

Character: `*` (star)
Description: Repeats the previous item zero or more times. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is not matched at all.
Example: «`".*"`» matches „`"def" "ghi"`” in “`abc "def" "ghi" jkl`”

Character: `*?` (lazy star)
Description: Repeats the previous item zero or more times. Lazy, so the engine first attempts to skip the previous item, before trying permutations with ever increasing matches of the preceding item.
Example: «`".*?"`» matches „`"def"`” in “`abc "def" "ghi" jkl`”

Character: `+` (plus)
Description: Repeats the previous item once or more. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is matched only once.
Example: «`".+"`» matches „`"def" "ghi"`” in “`abc "def" "ghi" jkl`”

Character: `+?` (lazy plus)
Description: Repeats the previous item once or more. Lazy, so the engine first matches the previous item only once, before trying permutations with ever increasing matches of the preceding item.
Example: «`".+?"`» matches „`"def"`” in “`abc "def" "ghi" jkl`”

Character: `{n}` where n is an integer >= 1
Description: Repeats the previous item exactly n times.
Example: «`a{3}`» matches „`aaa`”

Character: `{n,m}` where n >= 1 and m >= n
Description: Repeats the previous item between n and m times. Greedy, so repeating m times is tried before reducing the repetition to n times.
Example: «`a{2,4}`» matches „`aa`”, „`aaa`” or „`aaaa`”

Character: `{n,m}?` where n >= 1 and m >= n
Description: Repeats the previous item between n and m times. Lazy, so repeating n times is tried before increasing the repetition to m times.
Example: «`a{2,4}?`» matches „`aaaa`”, „`aaa`” or „`aa`”

Character: `{n,}` where n >= 1
Description: Repeats the previous item at least n times. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is matched only n times.
Example: «`a{2,}`» matches „`aaaaa`” in “`aaaaa`”

Character: `{n,}?` where n >= 1
Description: Repeats the previous item between n and m times. Lazy, so the engine first matches the previous item n times, before trying permutations with ever increasing matches of the preceding item.
Example: «`a{2,}?`» matches „`aa`” in “`aaaaa`”

2. Advanced Syntax Reference

Grouping and Backreferences

Character: `(regex)`
Description: Round brackets group the regex between them. They capture the text matched by the regex inside them that can be reused in a backreference, and they allow you to apply regex operators to the entire grouped regex.
Example: «`(abc){3}`» matches „`abcabcabc`”. First group matches „`abc`”.

Character: `(?:regex)`
Description: Non-capturing parentheses group the regex so you can apply regex operators, but do not capture anything and do not create backreferences.
Example: «`(?:abc){3}`» matches „`abcabcabc`”. No groups.

Character: `\1` through `\9`
Description: Substituted with the text matched between the 1st through 9th pair of capturing parentheses. Some regex flavors allow more than 9 backreferences.
Example: «`(abc|def)=\1`» matches „`abc=abc`” or „`def=def`”, but not “`abc=def`” or “`def=abc`”.

Modifiers

Character: `(?i)`
Description: Turn on case insensitivity for the remainder of the regular expression. (Older regex flavors may turn it on for the entire regex.)
Example: «`te(?i)st`» matches „`teST`” but not “`TEST`”.

Character: `(?-i)`
Description: Turn off case insensitivity for the remainder of the regular expression.
Example: «`(?i)te(?-i)st`» matches „`TEst`” but not “`TEST`”.

Character: `(?s)`
Description: Turn on "dot matches newline" for the remainder of the regular expression. (Older regex flavors may turn it on for the entire regex.)
Character: `(?-s)`
Description: Turn off "dot matches newline" for the remainder of the regular expression.
Character: `(?m)`
Description: Caret and dollar match after and before newlines for the remainder of the regular expression. (Older regex flavors may apply this to the entire regex.)
Character: `(?-m)`
Description: Caret and dollar only match at the start and end of the string for the remainder of the regular expression.
Character: `(?i-sm)`
Description: Turns on the options "i" and "m", and turns off "s" for the remainder of the regular expression. (Older regex flavors may apply this to the entire regex.)
Character: `(?i-sm:regex)`
Description: Matches the regex inside the span with the options "i" and "m" turned on, and "s" turned off.
Example: «`(?i:te)st`» matches „`TEst`” but not “`TEST`”.

Atomic Grouping and Possessive Quantifiers

Character: `(?>regex)`
Description: Atomic groups prevent the regex engine from backtracking back into the group (forcing the group to discard part of its match) after a match has been found for the group. Backtracking can occur inside the group before it has matched completely, and the engine can backtrack past the entire group, discarding its match entirely. Eliminating needless backtracking provides a speed increase. Atomic grouping is often indispensable when nesting quantifiers to prevent a catastrophic amount of backtracking as the engine needlessly tries pointless permutations of the nested quantifiers.
Example: «`x(?>\w+)x`» is more efficient than «`x\w+x`» if the second x cannot be matched.

Character: `?+`, `*+`, `++` and `{m,n}+`
Description: Possessive quantifiers are a limited yet syntactically cleaner alternative to atomic grouping. Only available in a few regex flavors. They behave as normal greedy quantifiers, except that they will not give up part of their match for backtracking.
Example: «`x++`» is identical to «`(?>x+)`»

Lookaround

Character: `(?=regex)`
Description: Zero-width positive lookahead. Matches at a position where the pattern inside the lookahead can be matched. Matches only the position. It does not consume any characters or expand the match. In a pattern like «`one(?=two)three`», both «`two`» and «`three`» have to match at the position where the match of «`one`» ends.
Example: «`t(?=s)`» matches the second „`t`” in „`streets`”.

Character: `(?!regex)`
Description: Zero-width negative lookahead. Identical to positive lookahead, except that the overall match will only succeed if the regex inside the lookahead fails to match.
Example: «`t(?!s)`» matches the first „`t`” in „`streets`”.

Character: `(?<=text)`
Description: Zero-width positive lookbehind. Matches at a position to the left of which text appears. Since regular expressions cannot be applied backwards, the test inside the lookbehind can only be plain text. Some regex flavors allow alternation of plain text options in the lookbehind.
Example: «`(?<=s)t`» matches the first „`t`” in „`streets`”.

Character: `(?<!text)`
Description: Zero-width negative lookbehind. Matches at a position if the text does not appear to the left of that position.
Example: «`(?<!s)t`» matches the second „`t`” in „`streets`”.

Continuing from The Previous Match

Character: `\G`
Description: Matches at the position where the previous match ended, or the position where the current match attempt started (depending on the tool or regex flavor). Matches at the start of the string during the first match attempt.
Example: «`\G[a-z]`» first matches „`a`”, then matches „`b`” and then fails to match in “`ab_cd`”.

Conditionals

Character: `(?(?=regex)then|else)`
Description: If the lookahead succeeds, the "then" part must match for the overall regex to match. If the lookahead fails, the "else" part must match for the overall regex to match. Not just positive lookahead, but all four lookarounds can be used. Note that the lookahead is zero-width, so the "then" and "else" parts need to match and consume the part of the text matched by the lookahead as well.
Example: «`(?(?<=a)b|c)`» matches the second „`b`” and the first „`c`” in “`babxcac`”

Comments

Character: `(?#comment)`
Description: Everything between `(?#` and `)` is ignored by the regex engine.
Example: «`a(?#foobar)b`» matches „`ab`”

3. Syntax Reference for Specific Regex Flavors

.NET Syntax for Named Capture and Backreferences

Character: `(?<name>regex)`
Description: Round brackets group the regex between them. They capture the text matched by the regex inside them that can be referenced by the name between the sharp brackets. The name may consist of letters and digits.
Character: `(?'name'regex)`
Description: Round brackets group the regex between them. They capture the text matched by the regex inside them that can be referenced by the name between the single quotes. The name may consist of letters and digits.
Character: `\k<name>`
Description: Substituted with the text matched by the capturing group with the given name.
Example: «`(?<group>abc|def)=\k<group>`» matches „`abc=abc`” or „`def=def`”, but not “`abc=def`” or “`def=abc`”.

Character: `\k'name'`
Description: Substituted with the text matched by the capturing group with the given name.
Example: «`(?'group'abc|def)=\k'group'`» matches „`abc=abc`” or „`def=def`”, but not “`abc=def`” or “`def=abc`”.

Python Syntax for Named Capture and Backreferences

Character: `(?P<name>regex)`
Description: Round brackets group the regex between them. They capture the text matched by the regex inside them that can be referenced by the name between the sharp brackets. The name may consist of letters and digits.
Character: `(?P=name)`
Description: Substituted with the text matched by the capturing group with the given name. Not a group, despite the syntax using round brackets.
Example: «`(?P<group>abc|def)=(?P=group)`» matches „`abc=abc`” or „`def=def`”, but not “`abc=def`” or “`def=abc`”.

4. Unicode Syntax Reference

Unicode Characters

Character: `\X`
Description: Matches a single Unicode grapheme, whether encoded as a single code point or multiple code points using combining marks. A grapheme most closely resembles the everyday concept of a "character".
Example: «`\X`» matches „à” encoded as U+0061 U+0300, „à” encoded as U+00E0, „©”, etc.

Character: `\uFFFF` where FFFF are 4 hexadecimal digits
Description: Matches a specific Unicode code point. Can be used inside character classes.
Example: «`\u00E0`» matches „à” encoded as U+00E0 only. «`\u00A9`» matches „©”

Unicode Properties

Character: `\p{L}` or `\p{Letter}`
Description: Matches a single Unicode code point that has the property "letter". See Unicode Character Properties in the tutorial for a complete list of properties. Each Unicode code point has exactly one property. Can be used inside character classes.
Example: «`\p{L}`» matches „à” encoded as U+00E0; «`\p{S}`» matches „©”

Character: `\P{L}` or `\P{Letter}`
Description: Matches a single Unicode code point that does *not* have the property "letter". Can be used inside character classes.
Example: «`\P{L}`» matches „©”

Index

#

$. *see* dollar sign
(. *see* round bracket
). *see* round bracket
*. *see* star
.. *see* dot
.rbg files, 30, 37
.rbl files, 28, 37
?. *see* question mark
[. *see* square bracket
\. *see* backslash
\1. *see* backreference
\a. *see* bell
\b. *see* word boundary
\d. *see* digit
\D. *see* digit
\e. *see* escape
\f. *see* form feed
\G. *see* previous match
\n. *see* line feed
\r. *see* carriage return
\s. *see* whitespace
\S. *see* whitespace
\t. *see* tab
\v. *see* vertical tab
\w. *see* word character
\W. *see* word character
]. *see* square bracket
^. *see* caret
{. *see* curly braces
|. *see* vertical bar
+. *see* plus
3rd party integration, 34

A

accessibility tools, 32
actions, 5
Add button, 25
alternation, 65
anchor, 60, 80, 86, 92
any character, 58
API, 35
application integration, 34
arguments
  command line, 34
array, 6, 21
ASCII, 52
assertion, 86
asterisk. *see* star
atomic grouping, 82
automatic save, 25
awk, 53

B

\b. *see* word boundary
backreference, 70
  .NET, 71
  EditPad Pro, 70
  in a character class, 73
  number, 71
  Perl, 71
  PowerGREP, 70
  repetition, 72
backslash, 51, 52
  in a character class, 55
backtracking, 68, 81
backup, 29
Basic, 23
begin file, 61
begin line, 60
begin string, 60
bell, 52
benchmark, 16
braces. *see* curly braces
bracket. *see* square bracket *or* parenthesis

C

C, 20, 23
C#, 20, 23, 34. *see* .NET
C/C++, 142
C++, 23, 34
canonical equivalence
  Java, 78, 125
capturing group, 70
caret, 51, 60, 80
  in a character class, 55
carriage return, 52
case insensitive, 80
  .NET, 136
  Java, 125
  Perl, 143
character class, 55
  negated, 55

negated shorthand, 57
repeating, 57
shorthand, 56
special characters, 55
character range, 55
character set. *see* character class
characters, 51
ASCII, 52
categories, 77
digit, 56
in a character class, 55
invisible, 52
metacharacters, 51
non-printable, 52
non-word, 56, 63
special, 51
Unicode, 52, 76
whitespace, 56
word, 56, 63
check, 12
choice, 65
class, 55
Clear button, 30
clipboard, 12, 23
closing bracket, 78
closing quote, 78
code editor, 19
code point, 76
code snippets, 19
collect information, 115
COM Automation interface, 41
combining character, 77
combining mark, 76
combining multiple regexes, 65
comma-delimited list, 6
command line, 114
command line examples, 34
command line parameters, 36
comments, 96
compatibility, 49
condition
if-then-else, 94
conditions
many in one regex, 90
configure RegexBuddy, 32
console, 114
continue
from previous match, 92
control characters, 52, 78
copy, 23
copy all searched files, 29
copy only modified files, 29
count matches, 13
Create tab, 7, 10, 11
cross. *see* plus
Ctrl+C, 23
Ctrl+V, 23
curly braces, 67
currency sign, 77
cursor, 32

D

\d. *see* digit
\D. *see* digit
dash, 78
data, 49
date, 106
Debug button, 14
Debug tab, 14
debugger, 14
default value for parameters, 27
delete backup files, 30
Delphi, 20, 23, 34, 142
details, 21
DFA engine, 53
digit, 56, 78
digits, 56
distance, 110
DLL, 142
documentation, 11
dollar, 80
dollar sign, 51, 60
dot, 51, 58, 80
misuse, 81
newlines, 58
vs. negated character class, 59
dot net. *see* .NET
double quote, 51
duplicate lines, 109

E

eager, 53, 65
ECMAScript, 130, 136
EditPad Pro, 50, 117
backreference, 70
group, 70
efficiency, 16
egrep, 53, 113
else, 94
email address, 102
enclosing mark, 77
end file, 61
end line, 60

end of line, 52
end string, 60
engine, 49, 53
entire string, 60
entirely matching, 21
ereg, 145
escape, 23, 51, 52
  in a character class, 55
example
  .NET, 135
  date, 106
  duplicate lines, 109
  exponential number, 101, 109
  floating point number, 101
  HTML tags, 99
  integer number, 109
  Java, 123
  keywords, 109
  not meeting a condition, 108
  number, 109
  prepend lines, 61
  quoted string, 59
  reserved words, 109
  scientific number, 101, 109
  trimming whitespace, 99
  VBScript, 155
  whole line, 108
examples
  command line, 34
exception handling, 19. *see* catch. *see* try
execute, 30
explain token, 10
export, 11
Export button, 30

F

file mask, 29
find, 5
find first, 13, 21
Find First button, 13
find next, 13
Find Next button, 13
flavor, 49
flex, 53
floating point number, 101
folders, 29
form feed, 52
full stop. *see* dot

G

GNU grep, 113
grapheme, 76
greedy, 66, 67
grep, 113
  multi-line, 115
  PowerGREP, 115
GREP button, 30
GREP tab, 29
group, 70
  .NET, 71
  atomic, 82
  capturing, 70
  EditPad Pro, 70
  in a character class, 73
  named, 74
  nested, 82
  once-only, 82
  Perl, 71
  PowerGREP, 70
  repetition, 72

H

handle exceptions, 19
highlight, 12
Highlight button, 12, 33
HTML file export, 11
HTML tags, 99
hyphen, 78
  in a character class, 55

I

IDE, 19
if-then-else, 94
implementation, 19
information
  collecting, 115
insert token, 7
integer number, 109
integration with other tools, 34
invert mask, 29
invert results, 29
invisible characters, 52

J

J#, 34
Java, 20, 23, 120, 123
  appendReplacement(), 128
  appendTail, 129
  canonical equivalence, 125
  case insensitive, 125
  compile(), 126

demo application, 123
dot all, 126
find(), 127
literal strings, 122
Matcher class, 121, 126
matcher(), 126
matches(), 124
multi-line, 126
Pattern class, 121, 126
replaceAll(), 124, 128
source code example, 123
split(), 125
String class, 120
java.util.regex, 120
JavaScript, 20, 24, 130
JDK 1.4, 120

K

keywords, 109

L

language
C/C++, 142
ECMAScript, 130
Java, 120
JavaScript, 130
Perl, 143
PHP, 145
Python, 147
Ruby, 150
VBScript, 152
languages, 19, 23
launch RegexBuddy, 34
lazy, 68
better alternative, 68
leftmost match, 53
letter, 77. *see* word character
lex, 53
libraries, 19
library, 25
parameters, 27
Library tab, 28
limit, 6
line, 60
begin, 60
duplicate, 109
end, 60
not meeting a condition, 108
prepend, 61
line break, 52, 80
line feed, 52
line separator, 77
line terminator, 52
line-based, 29
Linux grep, 113
Linux library, 142
list all, 13
List All button, 13
literal characers, 51
lookahead, 86
lookaround, 86
many conditions in one regex, 90
lookbehind, 87
limitations, 87
lowercase letter, 77

M

many conditions in one regex, 90
mark, 77
match, 5, 19, 49
match mode, 80
matching entirely, 21
mathematical symbol, 77
metacharacters, 51
in a character class, 55
Microsoft .NET. *see* .NET
mode modifier, 80
mode span, 80
modifier, 80
modifier span, 80
modify libraries, 25
multi .bak, 30
multi backup N of, 30
multi-line, 80
.NET, 136
Java, 126
multi-line grep, 115
multi-line mode, 60
multiple regexes combined, 65
MySQL, 53

N

.NET, 132, 135
backreference, 71
demo application, 135
ECMAScript, 136
group, 71
groups, 138
IgnoreCase, 136
IsMatch(), 136
Match object, 138
Match(), 137, 139

MultiLine, 136
NextMatch(), 140
Regex(), 139
RegexOptions, 136
Replace, 137
Replace(), 140
SingleLine, 136
source code example, 135
Split(), 138, 141
named group, 74
near, 110
negated character class, 55
negated shorthand, 57
negative lookahead, 86
negative lookbehind, 87
nested grouping, 82
newline, 58, 80
Next button, 13
NFA engine, 53
no backups, 29
non-printable characters, 52
non-spacing mark, 77
number, 56, 78, 109
backreference, 71
exponential, 101, 109
floating point, 101
scientific, 101, 109
number of matches, 13

O

object, 22
once or more, 67
once-only grouping, 82
Open button, 25, 28, 30
open file, 12
open libraries, 25, 28
opening bracket, 78
opening quote, 78
operator, 24
optimization, 16
option, 65, 66, 67
options, 32
or
one character or another, 55
one regex or another, 65

P

paragraph separator, 77
parameterize a regex, 27
Parameters tab, 27
parenthesis. *see* round bracket
Pascal, 23
paste, 23
paste subject, 12
pattern, 49
PCRE, 20, 142
period. *see* dot
Perl, 20, 143
backreference, 71
group, 71
PHP, 20, 24, 145
ereg, 145
preg, 146
split, 145
pipe symbol. *see* vertical bar
plus, 51, 67
possessive quantifiers, 82
positive lookahead, 86
positive lookbehind, 87
possessive, 82
PowerGREP, 115
backreference, 70
group, 70
precedence, 65, 70
preferences, 32
preg, 24, 146
prepend lines, 61
preview, 30
Previous button, 13
previous match, 92
Procmail, 53
programming
Java, 120
Perl, 143
programming languages, 19, 23
punctuation, 78
Python, 20, 24, 147

Q

quantifier
atomic, 82
backreference, 72
backtracking, 68
curly braces, 67
greedy, 67
group, 72
lazy, 68
nested, 82
once or more, 67
once-only, 82
plus, 67
possessive, 82

question mark, 66
specific amount, 67
star, 67
zero or more, 67
zero or once, 66
question mark, 51, 66
common mistake, 101
lazy quantifiers, 68
quick execute, 30
quote, 51
quoted string, 59

R

raise. *see* exception handling
range of characters, 55
rbg files, 30, 37
rbl files, 28, 37
read only, 25
regex engine, 53
regex structure, 10
regex template, 27
regex tool, 115
regex tree, 10, 11
regex variants, 27
regex with parameters, 27
RegexBuddy Library file, 25
RegexBuddy.rbl, 28
regex-directed engine, 53
regular expression, 49
repetition
atomic, 82
backreference, 72
backtracking, 68
curly braces, 67
greedy, 67
group, 72
lazy, 68
nested, 82
once or more, 67
once-only, 82
plus, 67
possessive, 82
question mark, 66
specific amount, 67
star, 67
zero or more, 67
zero or once, 66
replace, 5, 19, 21
replace all, 13
Replace All button, 13
replacement text, 70
requirements
many in one regex, 90
reserved characters, 51
reuse, 25
part of the match, 70
RFC 822, 103
round bracket, 51, 70
Ruby, 20, 24, 150

S

\s. *see* whitespace
\S. *see* whitespace
same file name, 30
save, 11
Save As button, 25
Save button, 30
save libraries, 25
save one file for each searched file, 29
save regexes, 25
save results into a single file, 29
sawtooth, 18, 84
screen reader, 32
search, 5
search and replace, 50, 115
preview, 115
text editor, 117
send, 23
separator, 77
several conditions in one regex, 90
share regexes, 25
shorthand character class, 56
negated, 57
single .~??, 29
single .bak, 29
single quote, 51
single-line, 80
single-line mode, 58
software integration, 34
source code, 19
space separator, 77
spacing combining mark, 77
special characters, 51
in a character class, 55
in programming languages, 51
specific amount, 67
split, 6, 13, 19, 21
Split button, 13
square bracket, 51, 55
star, 51, 67
common mistake, 101
start file, 61

start line, 60
start string, 60
statistics, 115
stopwatch, 16
store regexes, 25
string, 49
  begin, 60
  end, 60
  matching entirely, 60
  quoted, 59
structure, 10
surrogate, 78
symbol, 77
syntax coloring, 33, 118
System.Text.RegularExpressions, 132

T

tab, 52
target, 29
task, 19
teaching materials, 11
template regex, 27
terminal, 114
terminate lines, 52
test, 12
Test tab, 12
text, 49
text cursor, 32
text editor, 50, 117
text file export, 11
text-directed engine, 53
throw. *see* exception handling
titlecase letter, 77
token, 7
tool
  EditPad Pro, 117
  egrep, 113
  GNU grep, 113
  grep, 113
  Linux grep, 113
  PowerGREP, 115
  specialized regex tool, 115
  text editor, 117
tool integration, 34
transfer, 23
tree, 10
trimming whitespace, 99
tutorial, 49
type library, 41

U

underscore, 56
undo GREP, 30
Unicode, 76
  Java, 125
UNIX grep, 113
uppercase letter, 77
use, 19
Use button, 28
Use tab, 19

V

variants of regexes, 27
VB.NET, 20
VBScript, 20, 152, 155
  demo application, 155
  source code example, 155
verify, 12
vertical bar, 51, 65
vertical tab, 52
Visual Basic, 20, 23, 34
Visual Basic.NET. *see* .NET
Visual Studio, 34

W

\w. *see* word character
\W. *see* word character
whitespace, 56, 77, 99
whole line, 60, 108
whole word, 63
Windows DLL, 142
word, 63
word boundary, 63
word character, 56, 63
words
  keywords, 109
workflow, 34

Z

zero or more, 67
zero or once, 66
zero-length match, 61
zero-width, 60, 86