Regular Expressions

The Complete Tutorial

Jan Goyvaerts

Regular Expressions: The Complete Tutorial

Jan Goyvaerts

Last updated July 2007.

This book is published exclusively at http://www.regular-expressions.info/print.html

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information is provided on an “as is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Table of Contents

Tutorial

Introduction

A regular expression (regex or regexp for short) is a special text string for describing a search pattern. You can think of regular expressions as wildcards on steroids. You are probably familiar with wildcard notations such as *.txt to find all text files in a file manager. The regex equivalent is «.*\.txt» .

But you can do much more with regular expressions. In a text editor like EditPad Pro or a specialized text processing tool like PowerGREP, you could use the regular expression «\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b» to search for an email address. *Any* email address, to be exact. A very similar regular expression (replace the first \b with ^ and the last one with $) can be used by a programmer to check if the user entered a properly formatted email address. In just one line of code, whether that code is written in Perl, PHP, Java, a .NET language or a multitude of other languages.

Complete Regular Expression Tutorial

Do not worry if the above example or the quick start make little sense to you. Any non-trivial regex looks daunting to anybody not familiar with them. But with just a bit of experience, you will soon be able to craft your own regular expressions like you have never done anything else. The tutorial in this book explains everything bit by bit.

This tutorial is quite unique because it not only explains the regex syntax, but also describes in detail how the regex engine actually goes about its work. You will learn quite a lot, even if you have already been using regular expressions for some time. This will help you to understand quickly why a particular regex does not do what you initially expected, saving you lots of guesswork and head scratching when writing more complex regexes.

Applications & Languages That Support Regexes

There are many software applications and programming languages that support regular expressions. If you are a programmer, you can save yourself lots of time and effort. You can often accomplish with a single regular expression in one or a few lines of code what would otherwise take dozens or hundreds.

Not Only for Programmers

If you are not a programmer, you use regular expressions in many situations just as well. They will make finding information a lot easier. You can use them in powerful search and replace operations to quickly make changes across large numbers of files. A simple example is «gr[ae]y» which will find both spellings of the word grey in one operation, instead of two. There are many text editors and search and replace tools with decent regex support.

Part 1

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 between 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.

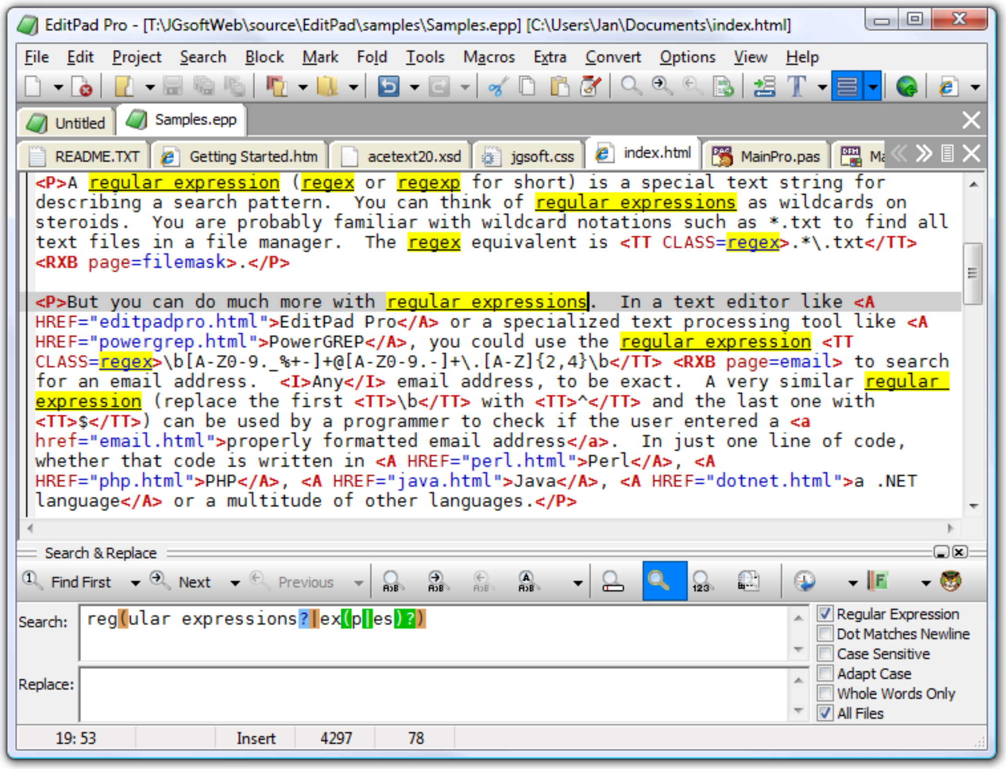
As usual in the software world, different regular expression engines are not fully compatible with each other. It is not possible to describe every kind of engine and regular expression syntax (or “flavor”) in this tutorial. I will focus on the regex flavor used by Perl 5, for the simple reason that 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 Search|Show Search Panel 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, and click the Find First button. This is the leftmost button on the search panel. See how EditPad Pro's regex engine finds the first match. Click the Find Next button, which sits next to the Find First button, to find further matches. When there are no further matches, the Find Next button's icon will flash briefly.

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. Select Count Matches in the Search menu to see how many times this regular expression can match the file you have open in 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.



```
<P>A regular expression (regex or regexp for short) is a special text string for
describing a search pattern.  You can think of regular expressions as wildcards on
steroids.  You are probably familiar with wildcard notations such as *.txt to find all
text files in a file manager.  The regex equivalent is <TT CLASS=regex>.*\.txt</TT>
<RXB page=filemask>.</P>

<P>But you can do much more with regular expressions.  In a text editor like <A
HREF="editpadpro.html">EditPad Pro</A> or a specialized text processing tool like <A
HREF="powergrep.html">PowerGREP</A>, you could use the regular expression <TT
CLASS=regex>\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b</TT> <RXB page=email> to search
for an email address.  <I>Any</I> email address, to be exact.  A very similar regular
expression (replace the first <TT>\b</TT> with <TT>^</TT> and the last one with
<TT>$</TT>) can be used by a programmer to check if the user entered a <a
href="email.html">properly formatted email address</a>.  In just one line of code,
whether that code is written in <A HREF="perl.html">Perl</A>, <A
HREF="php.html">PHP</A>, <A HREF="java.html">Java</A>, <A HREF="dotnet.html">a .NET
language</A> or a multitude of other languages.</P>
```

Search: `reg(ular expressions?|ex(p|es)?)`

Replace:

☑ Regular Expression
☐ Dot Matches Newline
☐ Case Sensitive
☐ Adapt Case
☐ Whole Words Only
☑ All Files

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.

Most regular expression flavors treat the brace «{» as a literal character, unless it is part of a repetition operator like «{1,3}». So you generally do not need to escape it with a backslash, though you can do so if you want. An exception to this rule is the java.util.regex package: it requires all literal braces to be escaped.

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.

Escaping a single metacharacter with a backslash works in all regular expression flavors. Many flavors also support the `\Q...\E` escape sequence. All the characters between the `\Q` and the `\E` are interpreted as literal characters. E.g. «`\Q*\d+*\E`» matches the literal text „`*\d+*`”. The `\E` may be omitted at the end of the regex, so «`\Q*\d+*`» is the same as «`\Q*\d+*\E`». This syntax is supported by the JGsoft engine, Perl and PCRE, both inside and outside character classes. Java supports it outside character classes only, and quantifies it as one token.

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.

Most regex flavors also support the tokens «`\cA`» through «`\cZ`» to insert ASCII control characters. The letter after the backslash is always a lowercase c. The second letter is an uppercase letter A through Z, to indicate Control+A through Control+Z. These are equivalent to «`\x01`» through «`\x1A`» (26 decimal). E.g. «`\cM`» matches a carriage return, just like «`\r`» and «`\x0D`». In XML Schema regular expressions, «`\c`» is a shorthand character class that matches any character allowed in an XML name.

If your regular expression engine supports Unicode, use «`\uFFFF`» rather than «`\xFF`» to insert a Unicode character. The euro currency sign occupies code point 0x20AC. If you cannot type it on your keyboard, you can insert it into a regular expression with «`\u20AC`».

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 all non-printable characters in character classes just like you can use them outside of character classes. E.g. «[$\u20AC]» matches a dollar or euro sign, assuming your regex flavor supports Unicode.

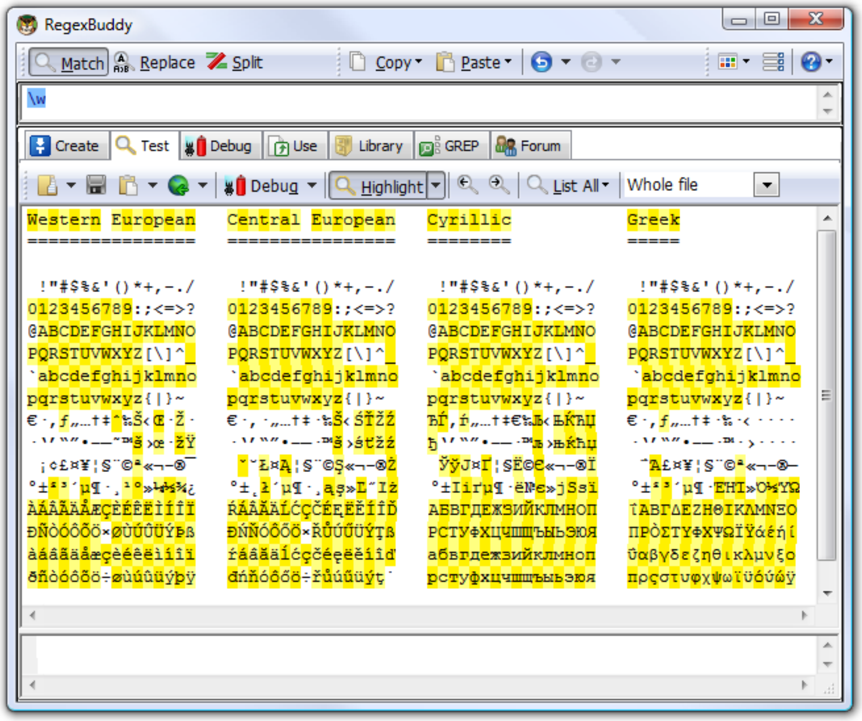
The JGsoft engine, Perl and PCRE also support the \Q...\E sequence inside character classes to escape a string of characters. E.g. «[\Q[-]\E]» matches „[”, „-” or „]”.

POSIX regular expressions treat the backslash as a literal character inside character classes. This means you can’t use backslashes to escape the closing bracket (]), the caret (^) and the hyphen (-). To use these characters, position them as explained above in this section. This also means that special tokens like shorthands are not available in POSIX regular expressions. See the tutorial topic on POSIX bracket expressions for more information.

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.



```
\w
```

```
Western European    Central European    Cyrillic            Greek
================    ================    ========            =====

 !"#$%&'()*+,-./     !"#$%&'()*+,-./     !"#$%&'()*+,-./     !"#$%&'()*+,-./
0123456789:;<=>?    0123456789:;<=>?    0123456789:;<=>?    0123456789:;<=>?
@ABCDEFGHIJKLMNO    @ABCDEFGHIJKLMNO    @ABCDEFGHIJKLMNO    @ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_    PQRSTUVWXYZ[\]^_    PQRSTUVWXYZ[\]^_    PQRSTUVWXYZ[\]^_
`abcdefghijklmno    `abcdefghijklmno    `abcdefghijklmno    `abcdefghijklmno
pqrstuvwxyz{|}~     pqrstuvwxyz{|}~     pqrstuvwxyz{|}~     pqrstuvwxyz{|}~
€·‚ƒ„…†‡ˆ‰Š‹Œ·Ž·    €·‚·„…†‡·‰Š‹ŚŤŽŹ    ЂЃ‚ѓ„…†‡€‰Љ‹ЊЌЋЏ    €·‚ƒ„…†‡·‰·‹····
·‘’“”•–—˜™š›œ·žŸ    ·‘’“”•–—·™š›śťžź    ђ‘’“”•–—·™љ›њќћџ    ·‘’“”•–—·™·›····
 ¡¢£¤¥¦§¨©ª«¬-®¯     ˇ˘Ł¤Ą¦§¨©Ş«¬-®Ż     ЎўЈ¤Ґ¦§Ё©Є«¬-®Ї     ΅Ά£¤¥¦§¨©·«¬-®―
°±²³´µ¶·¸¹º»¼½¾¿    °±˛ł´µ¶·¸ąş»Ľ˝ľż    °±Ііґµ¶·ё№є»јЅѕї    °±²³΄µ¶·ΈΉΊ»Ό½ΎΏ
ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏ    ŔÁÂĂÄĹĆÇČÉĘËĚÍÎĎ    АБВГДЕЖЗИЙКЛМНОП    ΐΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟ
ÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞß    ĐŃŇÓÔŐÖ×ŘŮÚŰÜÝŢß    РСТУФХЦЧШЩЪЫЬЭЮЯ    ΠΡ·ΣΤΥΦΧΨΩΪΫάέήί
àáâãäåæçèéêëìíîï    ŕáâăäĺćçčéęëěíîď    абвгдежзийклмноп    ΰαβγδεζηθικλμνξο
ðñòóôõö÷øùúûüýþÿ    đńňóôőö÷řůúűüýţ˙    рстуфхцчшщъыьэюя    πρςστυφχψωϊϋόύώ·
```

«\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”.

Tcl Word Boundaries

Word boundaries, as described above, are supported by all regular expression flavors described in in this book , except for the two POSIX RE flavors and the Tcl regexp command. POSIX does not support word boundaries at all. Tcl uses a different syntax.

In Tcl, «\b» matches a backspace character, just like «\x08» in most regex flavors (including Tcl’s). «\B» matches a single backslash character in Tcl, just like «\\» in all other regex flavors (and Tcl too).

Tcl uses the letter “y” instead of the letter “b” to match word boundaries. «\y» matches at any word boundary position, while «\Y» matches at any position that is not a word boundary. These Tcl regex tokens match exactly the same as «\b» and «\B» in Perl-style regex flavors. They don’t discriminate between the start and the end of a word.

Tcl has two more word boundary tokens that do discriminate between the start and end of a word. «\m» matches only at the start of a word. That is, it matches at any position that has a non-word character to the left of it, and a word character to the right of it. It also matches at the start of the string if the first character in the string is a word character. «\M» matches only at the end of a word. It matches at any position that has a word character to the left of it, and a non-word character to the right of it. It also matches at the end of the string if the last character in the string is a word character.

The only regex engine that supports Tcl-style word boundaries (besides Tcl itself) is the JGsoft engine. In PowerGREP and EditPad Pro, «\b» and «\B» are Perl-style word boundaries, and «\y», «\Y», «\m» and «\M» are Tcl-style word boundaries.

In most situations, the lack of «\m» and «\M» tokens is not a problem. «\yword\y» finds “whole words only” occurrences of “word” just like «\mword\M» would. «\Mword\m» could never match anywhere, since «\M» never matches at a position followed by a word character, and «\m» never at a position preceded by one. If your regular expression needs to match characters before or after «\y», you can easily specify in the regex whether these characters should be word characters or non-word characters. E.g. if you want to match any word, «\y\w+\y» will give the same result as «\m.+\M». Using «\w» instead of the dot automatically restricts the first «\y» to the start of a word, and the second «\y» to the end of a word. Note that «\y.+\y» would not work. This regex matches each word, and also each sequence of non-word characters between the words in your subject string. That said, if your flavor supports «\m» and «\M», the regex engine could apply «\m\w+\M» slightly faster than «\y\w+\y», depending on its internal optimizations.

If your regex flavor supports lookahead and lookbehind, you can use «(?<!\w)(?=\w)» to emulate Tcl's «\m» and «(?<=\w)(?!\w)» to emulate «\M». Though quite a bit more verbose, these lookaround constructs match exactly the same as Tcl's word boundaries.

If your flavor has lookahead but not lookbehind, and also has Perl-style word boundaries, you can use «\b(?=\w)» to emulate Tcl's «\m» and «\b(?!\w)» to emulate «\M». «\b» matches at the start or end of a word, and the lookahead checks if the next character is part of a word or not. If it is we're at the start of a word. Otherwise, we're at the end of a word.

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».

All regex flavors discussed in this book work this way, except one: the POSIX standard mandates that the longest match be returned, regardless if the regex engine is implemented using an NFA or DFA algorithm.

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. Lazy quantifiers are sometimes also called "ungreedy" or "reluctant". 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.

Repeating \Q...\E Escape Sequences

The \Q...\E sequence escapes a string of characters, matching them as literal characters. The JGsoft engine, Perl and PCRE treat the escaped characters as individual characters. If you place a quantifier after the \E, it will only be applied to the last character. E.g. if you apply «\Q*\d+*\E+» to "*\d+**\d+*", the match will be „*\d+**". Only the asterisk is repeated. (The plus repeats a token one or more times, as I'll explain later in this tutorial.) The Java engine, however, applies the quantifier to the whole \Q...\E sequence. So in Java, the above example matches the whole subject string „*\d+**\d+*".

If you want Java to return the same match as Perl, you'll need to split off the asterisk from the escape sequence, like this: «\Q*\d+\E*+». If you want Perl to repeat the whole sequence like Java does, simply group it: «(?:\Q*\d+*\E)+».

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, XML and the .NET framework use Unicode-based regex engines. Perl supports Unicode starting with version 5.6. PCRE can optionally be compiled with Unicode support. Note that PCRE is far less flexible in what it allows for the `\p` tokens, despite its name "Perl-compatible". The PHP preg functions, which are based on PCRE, support Unicode when the `/u` option is appended to the regular expression.

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. EditPad Pro supports Unicode starting with version 6.0.0.

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

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

All Unicode 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+».

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. You must always specify 4 hexadecimal digits E.g. «\u00E0» matches „à", but only when encoded as a single code point U+00E0.

Perl and PCRE do not support the «\uFFFF» syntax. They use «\x{FFFF}» instead. You can omit leading zeros in the hexadecimal number between the curly braces. Since \x by itself is not a valid regex token, «\x{1234}» can never be confused to match \x 1234 times. It always matches the Unicode code point U+1234. «\x{1234}{5678}» will try to match code point U+1234 exactly 5678 times.

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.

JavaScript, which does not offer any Unicode support through its RegExp class, does support «\uFFFF» for matching a single Unicode code point as part of its string syntax.

XML Schema does not have a regex token for matching Unicode code points. However, you can easily use XML entities like ￿ to insert literal code points into your regular expression.

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.

The .NET Regex class and PCRE are case sensitive when it checks the part between curly braces of a \p token. «\p{Zs}» will match any kind of space character, while «\p{zs}» will throw an error. All other regex engines described in this tutorial will match the space in both cases, ignoring the case of the property between the curly braces. Still, I recommend you make a habit of using the same uppercase and lowercase combination as I did in the list of properties below. This will make your regular expressions work with all Unicode regex engines.

In addition to the standard notation, «\p{L}», Java, Perl, PCRE and the JGsoft engine allow you to use the shorthand «\pL». The shorthand only works with single-letter Unicode properties. «\pLl» is *not* the equivalent of «\p{Ll}». It is the equivalent of «\p{L}l» which matches „Al" or „àl" or any Unicode letter followed by a literal „l".

Perl and the JGsoft engine also support the longhand «\p{Letter}». You can find a complete list of all Unicode properties below. You may omit the underscores or use hyphens or spaces instead.

- «\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.

Unicode Scripts

The Unicode standard places each assigned code point (character) into one script. A script is a group of code points used by a particular human writing system. Some scripts like Thai correspond with a single human language. Other scripts like Latin span multiple languages.

Some languages are composed of multiple scripts. There is no Japanese Unicode script. Instead, Unicode offers the Hiragana, Katakana, Han and Latin scripts that Japanese documents are usually composed of.

A special script is the Common script. This script contains all sorts of characters that are common to a wide range of scripts. It includes all sorts of punctuation, whitespace and miscellaneous symbols.

All assigned Unicode code points (those matched by «\P{Cn}») are part of exactly one Unicode script. All unassigned Unicode code points (those matched by «\p{Cn}») are not part of any Unicode script at all.

Very few regular expression engines support Unicode scripts today. Of all the flavors discussed in this tutorial, only the JGsoft engine, Perl and PCRE can match Unicode scripts. Here's a complete list of all Unicode scripts:

1. «\p{Common}»
2. «\p{Arabic}»
3. «\p{Armenian}»
4. «\p{Bengali}»
5. «\p{Bopomofo}»
6. «\p{Braille}»
7. «\p{Buhid}»
8. «\p{CanadianAboriginal}»
9. «\p{Cherokee}»
10. «\p{Cyrillic}»
11. «\p{Devanagari}»

12. «\p{Ethiopic}»
13. «\p{Georgian}»
14. «\p{Greek}»
15. «\p{Gujarati}»
16. «\p{Gurmukhi}»
17. «\p{Han}»
18. «\p{Hangul}»
19. «\p{Hanunoo}»
20. «\p{Hebrew}»
21. «\p{Hiragana}»
22. «\p{Inherited}»
23. «\p{Kannada}»
24. «\p{Katakana}»
25. «\p{Khmer}»
26. «\p{Lao}»
27. «\p{Latin}»
28. «\p{Limbu}»
29. «\p{Malayalam}»
30. «\p{Mongolian}»
31. «\p{Myanmar}»
32. «\p{Ogham}»
33. «\p{Oriya}»
34. «\p{Runic}»
35. «\p{Sinhala}»
36. «\p{Syriac}»
37. «\p{Tagalog}»
38. «\p{Tagbanwa}»
39. «\p{TaiLe}»
40. «\p{Tamil}»
41. «\p{Telugu}»
42. «\p{Thaana}»
43. «\p{Thai}»
44. «\p{Tibetan}»
45. «\p{Yi}»

Instead of the «\p{Latin}» syntax you can also use «\p{IsLatin}». The “Is” syntax is useful for distinguishing between scripts and blocks, as explained in the next section. Unfortunately, PCRE does not support “Is” as of this writing.

Unicode Blocks

The Unicode standard divides the Unicode character map into different blocks or ranges of code points. Each block is used to define characters of a particular script like “Tibetan” or belonging to a particular group like “Braille Patterns”. Most blocks include unassigned code points, reserved for future expansion of the Unicode standard.

Note that Unicode blocks do not correspond 100% with scripts. An essential difference between blocks and scripts is that a block is a single contiguous range of code points, as listed below. Scripts consist of characters taken from all over the Unicode character map. Blocks may include unassigned code points (i.e. code points matched by «`\p{Cn}`»). Scripts never include unassigned code points. Generally, if you're not sure whether to use a Unicode script or Unicode block, use the script.

E.g. the Currency block does not include the dollar and yen symbols. Those are found in the Basic_Latin and Latin-1_Supplement blocks instead, for historical reasons, even though both are currency symbols, and the yen symbol is not a Latin character. You should not blindly use any of the blocks listed below based on their names. Instead, look at the ranges of characters they actually match. A tool like RegexBuddy can be very helpful with this. E.g. the Unicode property «`\p{Sc}`» or «`\p{Currency_Symbol}`» would be a better choice than the Unicode block «`\p{InCurrency}`» when trying to find all currency symbols.

1. «`\p{InBasic_Latin}`»: U+0000..U+007F
2. «`\p{InLatin-1_Supplement}`»: U+0080..U+00FF
3. «`\p{InLatin_Extended-A}`»: U+0100..U+017F
4. «`\p{InLatin_Extended-B}`»: U+0180..U+024F
5. «`\p{InIPA_Extensions}`»: U+0250..U+02AF
6. «`\p{InSpacing_Modifier_Letters}`»: U+02B0..U+02FF
7. «`\p{InCombining_Diacritical_Marks}`»: U+0300..U+036F
8. «`\p{InGreek_and_Coptic}`»: U+0370..U+03FF
9. «`\p{InCyrillic}`»: U+0400..U+04FF
10. «`\p{InCyrillic_Supplementary}`»: U+0500..U+052F
11. «`\p{InArmenian}`»: U+0530..U+058F
12. «`\p{InHebrew}`»: U+0590..U+05FF
13. «`\p{InArabic}`»: U+0600..U+06FF
14. «`\p{InSyriac}`»: U+0700..U+074F
15. «`\p{InThaana}`»: U+0780..U+07BF
16. «`\p{InDevanagari}`»: U+0900..U+097F
17. «`\p{InBengali}`»: U+0980..U+09FF
18. «`\p{InGurmukhi}`»: U+0A00..U+0A7F
19. «`\p{InGujarati}`»: U+0A80..U+0AFF
20. «`\p{InOriya}`»: U+0B00..U+0B7F
21. «`\p{InTamil}`»: U+0B80..U+0BFF
22. «`\p{InTelugu}`»: U+0C00..U+0C7F
23. «`\p{InKannada}`»: U+0C80..U+0CFF
24. «`\p{InMalayalam}`»: U+0D00..U+0D7F
25. «`\p{InSinhala}`»: U+0D80..U+0DFF
26. «`\p{InThai}`»: U+0E00..U+0E7F
27. «`\p{InLao}`»: U+0E80..U+0EFF
28. «`\p{InTibetan}`»: U+0F00..U+0FFF
29. «`\p{InMyanmar}`»: U+1000..U+109F
30. «`\p{InGeorgian}`»: U+10A0..U+10FF
31. «`\p{InHangul_Jamo}`»: U+1100..U+11FF
32. «`\p{InEthiopic}`»: U+1200..U+137F
33. «`\p{InCherokee}`»: U+13A0..U+13FF
34. «`\p{InUnified_Canadian_Aboriginal_Syllabics}`»: U+1400..U+167F
35. «`\p{InOgham}`»: U+1680..U+169F
36. «`\p{InRunic}`»: U+16A0..U+16FF
37. «`\p{InTagalog}`»: U+1700..U+171F
38. «`\p{InHanunoo}`»: U+1720..U+173F
39. «`\p{InBuhid}`»: U+1740..U+175F
40. «`\p{InTagbanwa}`»: U+1760..U+177F
41. «`\p{InKhmer}`»: U+1780..U+17FF

42. «\p{InMongolian}»: U+1800..U+18AF
43. «\p{InLimbu}»: U+1900..U+194F
44. «\p{InTai_Le}»: U+1950..U+197F
45. «\p{InKhmer_Symbols}»: U+19E0..U+19FF
46. «\p{InPhonetic_Extensions}»: U+1D00..U+1D7F
47. «\p{InLatin_Extended_Additional}»: U+1E00..U+1EFF
48. «\p{InGreek_Extended}»: U+1F00..U+1FFF
49. «\p{InGeneral_Punctuation}»: U+2000..U+206F
50. «\p{InSuperscripts_and_Subscripts}»: U+2070..U+209F
51. «\p{InCurrency_Symbols}»: U+20A0..U+20CF
52. «\p{InCombining_Diacritical_Marks_for_Symbols}»: U+20D0..U+20FF
53. «\p{InLetterlike_Symbols}»: U+2100..U+214F
54. «\p{InNumber_Forms}»: U+2150..U+218F
55. «\p{InArrows}»: U+2190..U+21FF
56. «\p{InMathematical_Operators}»: U+2200..U+22FF
57. «\p{InMiscellaneous_Technical}»: U+2300..U+23FF
58. «\p{InControl_Pictures}»: U+2400..U+243F
59. «\p{InOptical_Character_Recognition}»: U+2440..U+245F
60. «\p{InEnclosed_Alphanumerics}»: U+2460..U+24FF
61. «\p{InBox_Drawing}»: U+2500..U+257F
62. «\p{InBlock_Elements}»: U+2580..U+259F
63. «\p{InGeometric_Shapes}»: U+25A0..U+25FF
64. «\p{InMiscellaneous_Symbols}»: U+2600..U+26FF
65. «\p{InDingbats}»: U+2700..U+27BF
66. «\p{InMiscellaneous_Mathematical_Symbols-A}»: U+27C0..U+27EF
67. «\p{InSupplemental_Arrows-A}»: U+27F0..U+27FF
68. «\p{InBraille_Patterns}»: U+2800..U+28FF
69. «\p{InSupplemental_Arrows-B}»: U+2900..U+297F
70. «\p{InMiscellaneous_Mathematical_Symbols-B}»: U+2980..U+29FF
71. «\p{InSupplemental_Mathematical_Operators}»: U+2A00..U+2AFF
72. «\p{InMiscellaneous_Symbols_and_Arrows}»: U+2B00..U+2BFF
73. «\p{InCJK_Radicals_Supplement}»: U+2E80..U+2EFF
74. «\p{InKangxi_Radicals}»: U+2F00..U+2FDF
75. «\p{InIdeographic_Description_Characters}»: U+2FF0..U+2FFF
76. «\p{InCJK_Symbols_and_Punctuation}»: U+3000..U+303F
77. «\p{InHiragana}»: U+3040..U+309F
78. «\p{InKatakana}»: U+30A0..U+30FF
79. «\p{InBopomofo}»: U+3100..U+312F
80. «\p{InHangul_Compatibility_Jamo}»: U+3130..U+318F
81. «\p{InKanbun}»: U+3190..U+319F
82. «\p{InBopomofo_Extended}»: U+31A0..U+31BF
83. «\p{InKatakana_Phonetic_Extensions}»: U+31F0..U+31FF
84. «\p{InEnclosed_CJK_Letters_and_Months}»: U+3200..U+32FF
85. «\p{InCJK_Compatibility}»: U+3300..U+33FF
86. «\p{InCJK_Unified_Ideographs_Extension_A}»: U+3400..U+4DBF
87. «\p{InYijing_Hexagram_Symbols}»: U+4DC0..U+4DFF
88. «\p{InCJK_Unified_Ideographs}»: U+4E00..U+9FFF
89. «\p{InYi_Syllables}»: U+A000..U+A48F
90. «\p{InYi_Radicals}»: U+A490..U+A4CF
91. «\p{InHangul_Syllables}»: U+AC00..U+D7AF
92. «\p{InHigh_Surrogates}»: U+D800..U+DB7F
93. «\p{InHigh_Private_Use_Surrogates}»: U+DB80..U+DBFF

94. «\p{InLow_Surrogates}»: U+DC00..U+DFFF
95. «\p{InPrivate_Use_Area}»: U+E000..U+F8FF
96. «\p{InCJK_Compatibility_Ideographs}»: U+F900..U+FAFF
97. «\p{InAlphabetic_Presentation_Forms}»: U+FB00..U+FB4F
98. «\p{InArabic_Presentation_Forms-A}»: U+FB50..U+FDFF
99. «\p{InVariation_Selectors}»: U+FE00..U+FE0F
100. «\p{InCombining_Half_Marks}»: U+FE20..U+FE2F
101. «\p{InCJK_Compatibility_Forms}»: U+FE30..U+FE4F
102. «\p{InSmall_Form_Variants}»: U+FE50..U+FE6F
103. «\p{InArabic_Presentation_Forms-B}»: U+FE70..U+FEFF
104. «\p{InHalfwidth_and_Fullwidth_Forms}»: U+FF00..U+FFEF
105. «\p{InSpecials}»: U+FFF0..U+FFFF

Not all Unicode regex engines use the same syntax to match Unicode blocks. Perl and use the «\p{InBlock}» syntax as listed above. .NET and XML use «\p{IsBlock}» instead. The JGsoft engine supports both notations. I recommend you use the "In" notation if your regex engine supports it. "In" can only be used for Unicode blocks, while "Is" can also be used for Unicode properties and scripts, depending on the regular expression flavor you're using. By using "In", it's obvious you're matching a block and not a similarly named property or script.

In .NET and XML, you must omit the underscores but keep the hyphens in the block names. E.g. Use «\p{IsLatinExtended-A}» instead of «\p{InLatin_Extended-A}». Perl and Java allow you to use an underscore, hyphen, space or nothing for each underscore or hyphen in the block's name. .NET and XML also compare the names case sensitively, while Perl and Java do not. «\p{islatinextended-a}» throws an error in .NET, while «\p{inlatinextended-a}» works fine in Perl and Java.

The JGsoft engine supports all of the above notations. You can use "In" or "Is", ignore differences in upper and lower case, and use spaces, underscores and hyphens as you like. This way you can keep using the syntax of your favorite programming language, and have it work as you'd expect in PowerGREP or EditPad Pro.

The actual names of the blocks are the same in all regular expression engines. The block names are defined in the Unicode standard. PCRE does not support Unicode blocks.

Alternative Unicode Regex Syntax

Unicode is a relatively new addition to the world of regular expressions. As you guessed from my explanations of different notations, different regex engine designers unfortunately have different ideas about the syntax to use. Perl and Java even support a few additional alternative notations that you may encounter in regular expressions created by others. I recommend against using these notations in your own regular expressions, to maintain clarity and compatibility with other regex flavors, and understandability by people more familiar with other flavors.

If you are just getting started with Unicode regular expressions, you may want to skip this section until later, to avoid confusion (if the above didn't confuse you already).

In Perl and PCRE regular expressions, you may encounter a Unicode property like «\p{^Lu}» or «\p{^Letter}». These are negated properties identical to «\P{Lu}» or «\P{Letter}». Since very few regex flavors support the «\p{^L}» notation, and all Unicode-compatible regex flavors (including Perl and PCRE) support «\P{L}», I strongly recommend you use the latter syntax.

Perl (but not PCRE) and Java support the «\p{IsL}» notation, prefixing one-letter and two-letter Unicode property notations with “Is”. Since very few regex flavors support the «\p{IsL}» notation, and all Unicode-compatible regex flavors (including Perl and Java) support «\p{L}», I strongly recommend you use the latter syntax.

Perl and Java allow you to omit the “In” when matching Unicode blocks, so you can write «\p{Arrows}» instead of «\p{InArrows}». Perl can also match Unicode scripts, and some scripts like “Hebrew” have the same name as a Unicode block. In that situation, Perl will match the Hebrew script instead of the Hebrew block when you write «\p{Hebrew}». While there are no Unicode properties with the same names as blocks, the property «\p{Currency_Symbol}» is confusingly similar to the block «\p{Currency}». As I explained in the section on Unicode blocks, the characters they match are quite different. To avoid all such confusion, I strongly recommend you use the “In” syntax for blocks, the “Is” syntax for scripts (if supported), and the shorthand syntax «\p{Lu}» for properties.

Again, the JGsoft engine supports all of the above oddball notations. This is only done to allow you to copy and paste regular expressions and have them work as they do in Perl or Java. You should consider these notations deprecated.

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.

14. Regex Matching Modes

Most regular expression engines discussed in this tutorial support the following four 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.
- `/x` enables “free-spacing mode”. In this mode, whitespace between regex tokens is ignored, and an unescaped # starts a comment.

Two languages that don’t support all of the above three are JavaScript and Ruby. Some regex flavors also have additional modes or options that have single letter equivalents. These are very implementation-dependent.

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. JavaScript and Python apply all mode modifiers to the entire regular expression. They don’t support the `(?-ismx)` syntax, since turning off an option is pointless when mode modifiers apply to the whole regular expressions. All options are off by default.

You can quickly test how the regex flavor you’re using handles mode modifiers. 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.

Modifier spans are supported by all regex flavors that allow you to use mode modifiers in the middle of the regular expression, and by those flavors only. These include the JGsoft engine, .NET, Java, Perl and PCRE.

15. 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.

Possessive quantifiers are a way to prevent the regex engine from trying all permutations. This is primarily useful for performance reasons. You can also use possessive quantifiers to eliminate certain matches.

How Possessive Quantifiers Work

Several modern regular expression flavors, including the JGsoft, Java and PCRE have a third kind of quantifier: the possessive quantifier. Like a greedy quantifier, a possessive quantifier will repeat the token as many times as possible. Unlike a greedy quantifier, it will *not* give up matches as the engine backtracks. With a possessive quantifier, the deal is all or nothing. You can make a quantifier possessive by placing an extra + after it. E.g. «*» is greedy, «*?» is lazy, and «*+» is possessive. «++», «?+» and «{n,m}+» are all possessive as well.

Let’s see what happens if we try to match «"[^"]*+"» against “"abc"”. The «"» matches the „"”. «[^"]» matches „a”, „b” and „c” as it is repeated by the star. The final «"» then matches the final „"” and we found an overall match. In this case, the end result is the same, whether we use a greedy or possessive quantifier. There is a slight performance increase though, because the possessive quantifier doesn’t have to remember any backtracking positions.

The performance increase can be significant in situations where the regex fails. If the subject is “"abc” (no closing quote), the above matching process will happen in the same way, except that the second «"» fails. When using a possessive quantifier, there are no steps to backtrack to. The regular expression does not have any alternation or non-possessive quantifiers that can give up part of their match to try a different permutation of the regular expression. So the match attempt fails immediately when the second «"» fails.

Had we used a greedy quantifier instead, the engine would have backtracked. After the «"» failed at the end of the string, the «[^"]*» would give up one match, leaving it with „ab”. The «"» would then fail to match “c”. «[^"]*» backtracks to just „a”, and «"» fails to match “b”. Finally, «[^"]*» backtracks to match zero characters, and «"» fails “a”. Only at this point have all backtracking positions been exhausted, and does the engine give up the match attempt. Essentially, this regex performs as many needless steps as there are characters following the unmatched opening quote.

When Possessive Quantifiers Matter

The main practical benefit of possessive quantifiers is to speed up your regular expression. In particular, possessive quantifiers allow your regex to fail faster. In the above example, when the closing quote fails to match, we *know* the regular expression couldn't have possibly skipped over a quote. So there's no need to backtrack and check for the quote. We make the regex engine aware of this by making the quantifier possessive. In fact, some engines, including the JGsoft engine detect that «[^"]*» and «"» are mutually exclusive when compiling your regular expression, and automatically make the star possessive.

Now, linear backtracking like a regex with a single quantifier does is pretty fast. It's unlikely you'll notice the speed difference. However, when you're nesting quantifiers, a possessive quantifier may save your day. Nesting quantifiers means that you have one or more repeated tokens inside a group, and the group is also repeated. That's when catastrophic backtracking often rears its ugly head. In such cases, you'll depend on possessive quantifiers and/or atomic grouping to save the day.

Possessive Quantifiers Can Change The Match Result

Using possessive quantifiers can change the result of a match attempt. Since no backtracking is done, and matches that would require a greedy quantifier to backtrack will not be found with a possessive quantifier. E.g. «".*"» will match „"abc"" in ""abc"x", but «".*+"» will not match this string at all.

In both regular expressions, the first «"» will match the first „"" in the string. The repeated dot then matches the remainder of the string „abc"x". The second «"» then fails to match at the end of the string.

Now, the paths of the two regular expressions diverge. The possessive dot-star wants it all. No backtracking is done. Since the «"» failed, there are no permutations left to try, and the overall match attempt fails. The greedy dot-star, while initially grabbing everything, is willing to give back. It will backtrack one character at a time. Backtracking to „abc"", «"» fails to match "x". Backtracking to „abc", «"» matches „"". An overall match „"abc"" was found.

Essentially, the lesson here is that when using possessive quantifiers, you need to make sure that whatever you're applying the possessive quantifier to should not be able to match what should follow it. The problem in the above example is that the dot also matches the closing quote. This prevents us from using a possessive quantifier. The negated character class in the previous section cannot match the closing quote, so we can make it possessive.

Using Atomic Grouping Instead of Possessive Quantifiers

Technically, possessive quantifiers are a notational convenience to place an atomic group around a single quantifier. All regex flavors that support possessive quantifiers also support atomic grouping. But not all regex flavors that support atomic grouping support possessive quantifiers. With those flavors, you can achieve the exact same results using an atomic group.

Basically, instead of «X*+», write «(>X*)». It is important to notice that both the quantified token X and the quantifier are inside the atomic group. Even if X is a group, you still need to put an extra atomic group around it to achieve the same effect. «(?:a|b)*+» is equivalent to «(?>(?:a|b)*)» but not to «(?>a|b)*».

The latter is a valid regular expression, but it won't have the same effect when used as part of a larger regular expression.

E.g. «(?:a|b)*+b» and «(?>(?:a|b)*)b» both fail to match "b". «a|b» will match the „b". The star is satisfied, and the fact that it's possessive or the atomic group will cause the star to forget all its backtracking positions. The second «b» in the regex has nothing left to match, and the overall match attempt fails.

In the regex «(?>a|b)*b», the atomic group forces the alternation to give up its backtracking positions. I.e. if an „a" is matched, it won't come back to try «b» if the rest of the regex fails. Since the star is outside of the group, it is a normal, greedy star. When the second «b» fails, the greedy star will backtrack to zero iterations. Then, the second «b» matches the „b" in the subject string.

This distinction is particularly important when converting a regular expression written by somebody else using possessive quantifiers to a regex flavor that doesn't have possessive quantifiers. You could, of course, let a tool like RegexBuddy do the job for you.

16. Atomic Grouping

An atomic group is a group that, when the regex engine exits from it, automatically throws away all backtracking positions remembered by any tokens inside the group. Atomic groups are non-capturing. The syntax is «(?>group)». Lookaround groups are also atomic. Atomic grouping is supported by most modern regular expression flavors, including the JGsoft flavor, Java, PCRE, .NET, Perl and Ruby. The first three of these also support possessive quantifiers, which are essentially a notational convenience for atomic grouping.

An example will make the behavior of atomic groups. The regular expression «a(bc|b)c» (capturing group) matches „abcc" and „abc". The regex «a(?>bc|b)c» (atomic group) matches „abcc" but not "abc".

When applied to "abc", both regexes will match «a» to „a", «bc» to „bc", and then «c» will fail to match at the end of the string. Here there paths diverge. The regex with the capturing group has remembered a backtracking position for the alternation. The group will give up its match, «b» then matches „b" and «c» matches „c". Match found!

The regex with the atomic group, however, exited from an atomic group after «bc» was matched. At that point, all backtracking positions for tokens inside the group are discarded. In this example, the alternation's option to try «b» at the second position in the string is discarded. As a result, when «c» fails, the regex engine has no alternatives left to try.

Of course, the above example isn't very useful. But it does illustrate very clearly how atomic grouping eliminates certain matches. Or more importantly, it eliminates certain match attempts.

Regex Optimization Using Atomic Grouping

Consider the regex «\b(integer|insert|in)\b» and the subject "integers". Obviously, because of the word boundaries, these don't match. What's not so obvious is that the regex engine will spend quite some effort figuring this out.

«\b» matches at the start of the string, and «integer» matches „integer". The regex engine makes note that there are to more alternatives in the group, and continues with «\b». This fails to match between the "r" and "s". So the engine backtracks to try the second alternative inside the group. The second alternative matches „in", but then fails to match «s». So the engine backtracks once more to the third alternative. «in» matches „in". «\b» fails between the "n" and "t" this time. The regex engine has no more remembered backtracking positions, so it declares failure.

This is quite a lot of work to figure out "integers" isn't in our list of words. We can optimize this by telling the regular expression engine that if it can't match «\b» after it matched „integer", then it shouldn't bother trying any of the other words. The word we've encountered in the subject string is a longer word, and it isn't in our list.

We can do this my turning the capturing group into an atomic group: «\b(?>integer|insert|in)\b». Now, when «integer» matches, the engine exits from an atomic group, and throws away the backtracking positions it stored for the alternation. When «\b» fails, the engine gives up immediately. This savings can be significant when scanning a large file for a long list of keywords. This savings will be vital when your alternatives contain repeated tokens (not to mention repeated groups) that lead to catastrophic backtracking.

Don't be too quick to make all your groups atomic. As we saw in the first example above, atomic grouping can exclude valid matches too. Compare how «\b(?>integer|insert|in)\b» and «\b(?>in|integer|insert)\b» behave when applied to "insert". The former regex matches, while the latter fails. If the groups weren't atomic, both regexes would match. Remember that alternation tries its alternatives from left to right. If the second regex matches „in", it won't try the two other alternatives due to the atomic group.

17. 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.

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 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, like PCRE and Java support the above, plus alternation with strings of different lengths. Each part of the alternation must still have a finite maximum length. 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. Unfortunately, the JDK 1.4 and 1.5 have some bugs when you use alternation inside lookbehind. These were fixed in JDK 1.6.

The only regex engines that allow you to use a full regular expression inside lookbehind are the JGsoft engine and the .NET framework RegEx classes.

Finally, flavors like JavaScript, Ruby and Tcl do not support lookbehind at all, even though they do support lookahead.

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.

18. Testing The Same Part of a 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.

19. 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 Languages

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.

The «\G» token is supported by the JGsoft engine, .NET, Java, Perl and PCRE.

20. 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 «(?».

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.

Looking Inside the Regex Engine

The regex «(a)?b(?(1)c|d)» matches „bd" and „abc". It does not match "bc", but does match „bd" in "abd". Let's see how this regular expression works on each of these four subject strings.

When applied to "bd", «a» fails to match. Since the capturing group containing «a» is optional, the engine continues with «b» at the start of the subject string. Since the whole group was optional, the group did not take part in the match. Any subsequent backreference to it like «\1» will fail. Note that «(a)?» is very different from «(a?)». In the former regex, the capturing group does not take part in the match if «a» fails, and backreferences to the group will fail. In the latter group, the capturing group always takes part in the match, capturing either „a" or nothing. Backreferences to a capturing group that took part in the match and captured nothing always succeed. Conditionals evaluating such groups execute the "then" part. In short: if you want to use a reference to a group in a conditional, use «(a)?» instead of «(a?)».

Continuing with our regex, «b» matches „b". The regex engine now evaluates the conditional. The first capturing group did not take part in the match at all, so the "else" part or «d» is attempted. «d» matches „d" and an overall match is found.

Moving on to our second subject string "abc", «a» matches „a", which is captured by the capturing group. Subsequently, «b» matches „b". The regex engine again evaluates the conditional. The capturing group took part in the match, so the "then" part or «c» is attempted. «c» matches „c" and an overall match is found.

Our third subject “bc” does not start with “a”, so the capturing group does not take part in the match attempt, like we saw with the first subject string. «b» still matches „b”, and the engine moves on to the conditional. The first capturing group did not take part in the match at all, so the “else” part or «d» is attempted. «d» does not match “c” and the match attempt at the start of the string fails. The engine does try again starting at the second character in the string, but fails since «b» does not match “c”.

The fourth subject “abd” is the most interesting one. Like in the second string, the capturing group grabs the „a” and the «b» matches. The capturing group took part in the match, so the “then” part or «c» is attempted. «c» fails to match “d”, and the match attempt fails. Note that the “else” part is not attempted at this point. The capturing group took part in the match, so only the “then” part is used. However, the regex engine isn’t done yet. It will restart the regular expression from the beginning, moving ahead one character in the subject string.

Starting at the second character in the string, «a» fails to match “b”. The capturing group does not take part in the second match attempt which started at the second character in the string. The regex engine moves beyond the optional group, and attempts «b», which matches. The regex engine now arrives at the conditional in the regex, and at the third character in the subject string. The first capturing group did not take part in the current match attempt, so the “else” part or «d» is attempted. «d» matches „d” and an overall match „bd” is found.

If you want to avoid this last match result, you need to use anchors. «^(a)?b(?(1)c|d)$» does not find any matches in the last subject string. The caret will fail to match at the second and third characters in the string.

Regex Flavors

Conditionals are supported by the JGsoft engine, Perl, PCRE and the .NET framework. All these flavors, except Perl, also support named capturing groups. They allow you to use the name of a capturing group instead of its number as the *if* test, e.g.: «(?<test>a)?b(?(test)c|d)».

Python supports conditionals using a numbered or named capturing group. Python does not support conditionals using lookaround, even though Python does support lookaround outside conditionals. Instead of a conditional like «(?(?=regex)then|else)», you can alternate two opposite lookarounds: «(?=regex)then|(?!regex)else)».

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. This simply matches the remainder of the line, allowing for any test subject.

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. When implementing this regular expression, the first capturing group will store the name of the header ("From", "To", or "Subject"), and the third capturing group will store the value of the header.

You could try to match even more headers by putting another conditional into the "else" part. E.g. «^((From|To)|(Date)|Subject): ((?(2)\w+@\w+\.[a-z]+|(?(3)mm/dd/yyyy|.+))» would match a "From", "To", "Date" or "Subject", and use the regex «mm/dd/yyyy» to check if the date is valid. Obviously, the date validation regex is just a dummy to keep the example simple. The header is captured in the first group, and its validated contents in the fourth group.

As you can see, regular expressions using conditionals quickly become unwieldy. I recommend that you only use them if one regular expression is all your tool allows you to use. When programming, you're far better of using the regex «^(From|To|Date|Subject): (.+)» to capture one header with its unvalidated contents. In your source code, check the name of the header returned in the first capturing group, and then use a second regular expression to validate the contents of the header returned in the second capturing group of the first regex. Though you'll have to write a few lines of extra code, this code will be much easier to understand maintain. If you precompile all the regular expressions, using multiple regular expressions will be just as fast, if not faster, and the one big regex stuffed with conditionals.

21. XML Schema Character Classes

XML Schema Regular Expressions support the usual six shorthand character classes, plus four more. These four aren't supported by any other regular expression flavor. «\i» matches any character that may be the first character of an XML name, i.e. «[_:A-Za-z]». «\c» matches any character that may occur after the first character in an XML name, i.e. «[-._:A-Za-z0-9]». «\I» and «\C» are the respective negated shorthands. Note that the «\c» shorthand syntax conflicts with the control character syntax used in many other regex flavors.

You can use these four shorthands both inside and outside character classes using the bracket notation. They're very useful for validating XML references and values in your XML schemas. The regular expression «\i\c*» matches an XML name like „xml:schema". In other regular expression flavors, you'd have to spell this out as «[_:A-Za-z][-._:A-Za-z0-9]*». The latter regex also works with XML's regular expression flavor. It just takes more time to type in.

The regex «<\i\c*\s*>» matches an opening XML tag without any attributes. «</\i\c*\s*>» matches any closing tag. «<\i\c*(\s+\i\c*\s*=\s*("[^"]*"|'[^']*'))*\s*>» matches an opening tag with any number of attributes. Putting it all together, «<(\i\c*(\s+\i\c*\s*=\s*("[^"]*"|'[^']*'))*|/\i\c*)\s*>» matches either an opening tag with attributes or a closing tag.

Character Class Subtraction

While the regex flavor it defines is quite limited, the XML Schema adds a new regular expression feature not previously seen in any (popular) regular expression flavor: character class subtraction. Currently, this feature is only supported by the JGsoft and .NET regex engines (in addition to those implementing the XML Schema standard).

Character class subtraction makes it easy to match any single character present in one list (the character class), but not present in another list (the subtracted class). The syntax for this is [class-[subtract]]. If the character after a hyphen is an opening bracket, XML regular expressions interpret the hyphen as the subtraction operator rather than the range operator. E.g. «[a-z-[aeiuo]]» matches a single letter that is not a vowel (i.e. a single consonant). Without the character class subtraction feature, the only way to do this would be to list all consonants: «[b-df-hj-np-tv-z]».

This feature is more than just a notational convenience, though. You can use the full character class syntax within the subtracted character class. E.g. to match all Unicode letters except ASCII letters (i.e. all non-English letters), you could easily use «[\p{L}-[\p{IsBasicLatin}]]».

Nested Character Class Subtraction

Since you can use the full character class syntax within the subtracted character class, you can subtract a class from the class being subtracted. E.g. «[0-9-[0-6-[0-3]]]» first subtracts 0-3 from 0-6, yielding «[0-9-[4-6]]», or «[0-37-9]», which matches any character in the string "0123789".

The class subtraction must always be the last element in the character class. [0-9-[4-6]a-f] is not a valid regular expression. It should be rewritten as «[0-9a-f-[4-6]]». The subtraction works on the whole class.

E.g. «[\p{Ll}\p{Lu}-[\p{IsBasicLatin}]]» matches all uppercase and lowercase Unicode letters, except any ASCII letters. The \p{IsBasicLatin} is subtracted from the combination of \p{Ll}\p{Lu} rather than from \p{Lu} alone. This regex will not match "abc".

While you can use nested character class subtraction, you cannot subtract two classes sequentially. To subtract ASCII letters and Greek letters from a class with all Unicode letters, combine the ASCII and Greek letters into one class, and subtract that, as in «[\p{L}-[\p{IsBasicLatin}\p{IsGreek}]]».

Notational Compatibility with Other Regex Flavors

Note that a regex like «[a-z-[aeiuo]]» will not cause any errors in regex flavors that do not support character class subtraction. But it won't match what you intended either. E.g. in Perl, this regex consists of a character class followed by a literal «]». The character class matches a character that is either in the range a-z, or a hyphen, or an opening bracket, or a vowel. Since the a-z range and the vowels are redundant, you could write this character class as «[a-z-[]» or «[-[a-z]». A hyphen after a range is treated as a literal character, just like a hyphen immediately after the opening bracket. This is true in all regex flavors, including XML. E.g. «[a-z-_]» matches a lowercase letter, a hyphen or an underscore in both Perl and XML Schema.

While the last paragraph strictly speaking means that the XML Schema character class syntax is incompatible with Perl and the majority of other regex flavors, in practice there's no difference. Using non-alphanumeric characters in character class ranges is very bad practice, as it relies on the order of characters in the ASCII character table, which makes the regular expression hard to understand for the programmer who inherits your work. E.g. while «[A-[]» would match any upper case letter or an opening square bracket in Perl, this regex is much clearer when written as «[A-Z[]». The former regex would cause an error in XML Schema, because it interprets -[] as an empty subtracted class, leaving an unbalanced [.

22. POSIX Bracket Expressions

POSIX bracket expressions are a special kind of character classes. POSIX bracket expressions match one character out of a set of characters, just like regular character classes. The main purpose of the bracket expressions is that they adapt to the user's or application's locale. A locale is a collection of rules and settings that describe language and cultural conventions, like sort order, date format, etc. The POSIX standard also defines these locales.

Generally, only POSIX-compliant regular expression engines have proper and full support for POSIX bracket expressions. Some non-POSIX regex engines support POSIX character classes, but usually don't support collating sequences and character equivalents. Regular expression engines that support Unicode use Unicode properties and scripts to provide functionality similar to POSIX bracket expressions. In Unicode regex engines, shorthand character classes like «`\w`» normally match all relevant Unicode characters, alleviating the need to use locales.

Character Classes

Don't confuse the POSIX term "character class" with what is normally called a regular expression character class. «`[x-z0-9]`» is an example of what we call a "character class" and POSIX calls a "bracket expression". `[:digit:]` is a POSIX character class, used inside a bracket expression like «`[x-z[:digit:]]`». These two regular expressions match exactly the same: a single character that is either „`x`", „`y`", „`z`" or a digit. The class names must be written all lowercase.

POSIX bracket expressions can be negated. «`[^x-z[:digit:]]`» matches a single character that is not x, y, z or a digit. A major difference between POSIX bracket expressions and the character classes in other regex flavors is that POSIX bracket expressions treat the backslash as a literal character. This means you can't use backslashes to escape the closing bracket (]), the caret (^) and the hyphen (-). 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.

Exactly which POSIX character classes are available depends on the POSIX locale. The following are usually supported, often also by regex engines that don't support POSIX itself. I've also indicated equivalent character classes that you can use in ASCII and Unicode regular expressions if the POSIX classes are unavailable. Some classes also have Perl-style shorthand equivalents.

Java does not support POSIX bracket expressions, but does support POSIX character classes using the `\p` operator. Though the `\p` syntax is borrowed from the syntax for Unicode properties, the POSIX classes in Java only match ASCII characters as indicated below. The class names are case sensitive. Unlike the POSIX syntax which can only be used inside a bracket expression, Java's `\p` can be used inside and outside bracket expressions.

POSIX: «[:alnum:]»
Description: Alphanumeric characters
ASCII: «[a-zA-Z0-9]»
Unicode: «[\p{L&}\p{Nd}]»
Shorthand:
Java: «\p{Alnum}»

POSIX: «[:alpha:]»
Description: Alphabetic characters
ASCII: «[a-zA-Z]»
Unicode: «\p{L&}»
Shorthand:
Java: «\p{Alpha}»

POSIX: «[:ascii:]»
Description: ASCII characters
ASCII: «[\x00-\x7F]»
Unicode: «\p{InBasicLatin}»
Shorthand:
Java: «\p{ASCII}»

POSIX: «[:blank:]»
Description: Space and tab
ASCII: «[\t]»
Unicode: «[\p{Zs}\t]»
Shorthand:
Java: «\p{Blank}»

POSIX: «[:cntrl:]»
Description: Control characters
ASCII: «[\x00-\x1F\x7F]»
Unicode: «\p{Cc}»
Shorthand:
Java: «\p{Cntrl}»

POSIX: «[:digit:]»
Description: Digits
ASCII: «[0-9]»
Unicode: «\p{Nd}»
Shorthand: «\d»
Java: «\p{Digit}»

POSIX: «[:graph:]»
Description: Visible characters (i.e. anything except spaces, control characters, etc.)
ASCII: «[\x21-\x7E]»
Unicode: «[^\p{Z}\p{C}]»
Shorthand:
Java: «\p{Graph}»

POSIX: «[:lower:]»
Description: Lowercase letters
ASCII: «[a-z]»
Unicode: «\p{Ll}»
Shorthand:
Java: «\p{Lower}»

POSIX: «[:print:]»
Description: Visible characters and spaces (i.e. anything except control characters, etc.)
ASCII: «[\x20-\x7E]»
Unicode: «\P{C}»
Shorthand:
Java: «\p{Print}»

POSIX: «[:punct:]»
Description: Punctuation characters.
ASCII: «[!"#$%&'()*+,-./:;?@[\\\]_`{|}~]»
Unicode: «\p{P}»
Shorthand:
Java: «\p{Punct}»

POSIX: «[:space:]»
Description: All whitespace characters, including line breaks
ASCII: «[\t\r\n\v\f]»
Unicode: «[\p{Z}\t\r\n\v\f]»
Shorthand: «\s»
Java: «\p{Space}»

POSIX: «[:upper:]»
Description: Uppercase letters
ASCII: «[A-Z]»
Unicode: «\p{Lu}»
Shorthand:
Java: «\p{Upper}»

POSIX: «[:word:]»
Description: Word characters (letters, numbers and underscores)
ASCII: «[A-Za-z0-9_]»
Unicode: «[\p{L}\p{N}\p{Pc}]»
Shorthand: «\w»
Java:

POSIX: «[:xdigit:]»
Description: Hexadecimal digits
ASCII: «[A-Fa-f0-9]»
Unicode: «[A-Fa-f0-9]»
Shorthand:
Java: «\p{XDigit}»

Collating Sequences

A POSIX locale can have collating sequences to describe how certain characters or groups of characters should be ordered. E.g. in Spanish, “ll” like in “tortilla” is treated as one character, and is ordered between “l” and “m” in the alphabet. You can use the collating sequence element [.span-ll.] inside a bracket expression to match „ll”. E.g. the regex «torti[[.span-ll.]]a» matches „tortilla”. Notice the double square brackets. One pair for the bracket expression, and one pair for the collating sequence.

I do not know of any regular expression engine that support collating sequences, other than POSIX-compliant engines part of a POSIX-compliant system.

Note that a fully POSIX-compliant regex engine will treat “ll” as a single character when the locale is set to Spanish. This means that «torti[^x]a» also matches „tortilla”. «[^x]» matches a single character that is not an “x”, which includes „ll” in the Spanish POSIX locale.

In any other regular expression engine, or in a POSIX engine not using the Spanish locale, «torti[^x]a» will match the misspelled word „tortila” but will not match „tortilla”, as «[^x]» cannot match the two characters “ll”.

Finally, note that not all regex engines claiming to implement POSIX regular expressions actually have full support for collating sequences. Sometimes, these engines use the regular expression syntax defined by POSIX, but don’t have full locale support. You may want to try the above matches to see if the engine you’re using does. E.g. Tcl’s regexp command supports collating sequences, but Tcl only supports the Unicode locale, which does not define any collating sequences. The result is that in Tcl, a collating sequence specifying a single character will match just that character, and all other collating sequences will result in an error.

Character Equivalents

A POSIX locale can define character equivalents that indicate that certain characters should be considered as identical for sorting. E.g. in French, accents are ignored when ordering words. “élève” comes before “être” which comes before “événement”. “é” and “ê” are all the same as “e”, but “l” comes before “t” which comes before “v”. With the locale set to French, a POSIX-compliant regular expression engine will match „e”, „é”, „è” and „ê” when you use the collating sequence [=e=] in the bracket expression «[[=e=]]».

If a character does not have any equivalents, the character equivalence token simply reverts to the character itself. E.g. «[[=x=][=z=]]» is the same as «[xz]» in the French locale.

Like collating sequences, POSIX character equivalents are not available in any regex engine that I know of, other than those following the POSIX standard. And those that do may not have the necessary POSIX locale support. Here too Tcl’s regexp command supports character equivalents, but Unicode locale, the only one Tcl supports, does not define any character equivalents. This effectively means that «[[=x=]]» and «[x]» are exactly the same in Tcl, and will only match „x”, for any character you may try instead of “x”.

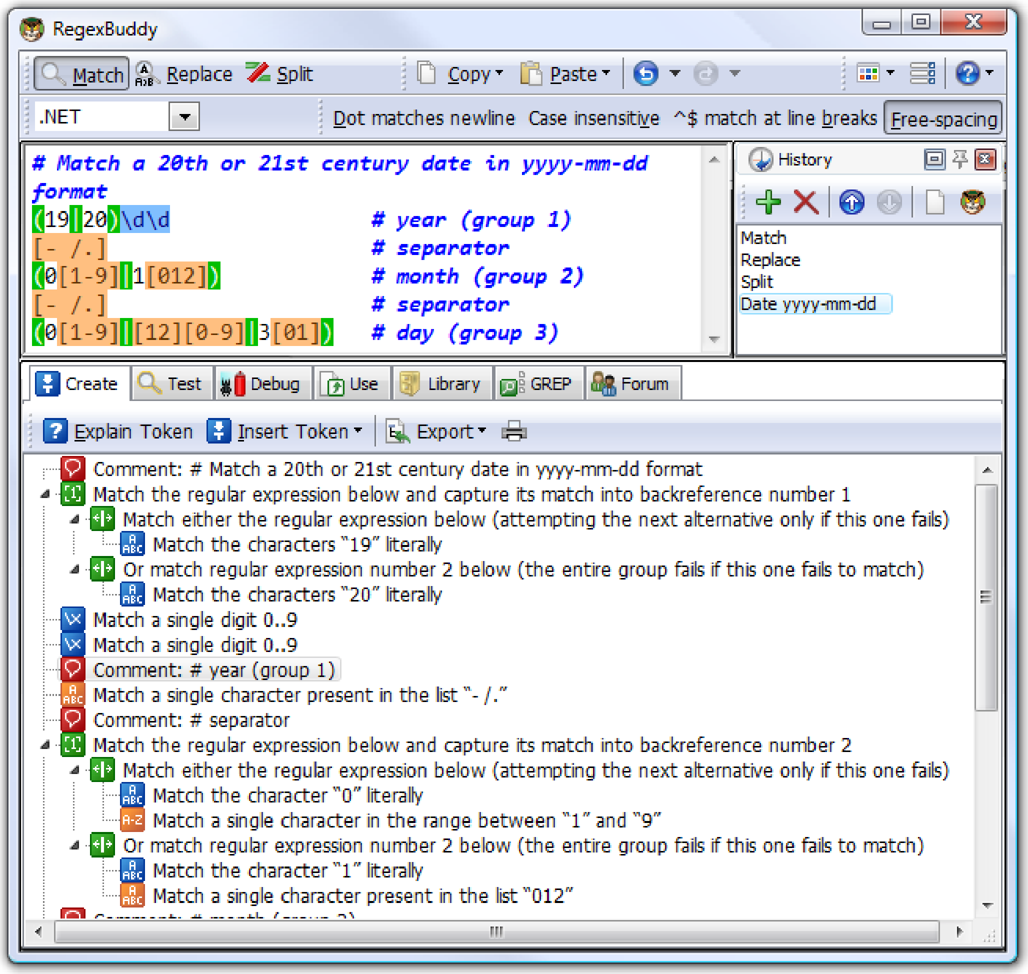
23. Adding Comments to Regular Expressions

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.

Regex comments are supported by the JGsoft engine, .NET, Perl, PCRE, Python and Ruby.

To make your regular expression even more readable, you can turn on free-spacing mode. All flavors that support comments also support free-spacing mode. In addition, Java supports free-spacing mode, even though it doesn't support (?#)-style comments.



```
# Match a 20th or 21st century date in yyyy-mm-dd
format
(19|20)\d\d               # year (group 1)
[- /.]                    # separator
(0[1-9]|1[012])           # month (group 2)
[- /.]                    # separator
(0[1-9]|[12][0-9]|3[01])  # day (group 3)
```

Match
Replace
Split
Date yyyy-mm-dd

- Comment: # Match a 20th or 21st century date in yyyy-mm-dd format
- Match the regular expression below and capture its match into backreference number 1
  - Match either the regular expression below (attempting the next alternative only if this one fails)
    - Match the characters "19" literally
  - Or match regular expression number 2 below (the entire group fails if this one fails to match)
    - Match the characters "20" literally
- Match a single digit 0..9
- Match a single digit 0..9
- Comment: # year (group 1)
- Match a single character present in the list "- /."
- Comment: # separator
- Match the regular expression below and capture its match into backreference number 2
  - Match either the regular expression below (attempting the next alternative only if this one fails)
    - Match the character "0" literally
    - Match a single character in the range between "1" and "9"
  - Or match regular expression number 2 below (the entire group fails if this one fails to match)
    - Match the character "1" literally
    - Match a single character present in the list "012"
- Comment: # month (group 2)

24. Free-Spacing Regular Expressions

The JGsoft engine, .NET, Java, Perl, PCRE, Python and Ruby support a variant of the regular expression syntax called free-spacing mode. You can turn on this mode with the «(?x)» mode modifier, or by turning on the corresponding option in the application or passing it to the regex constructor in your programming language.

In free-spacing mode, whitespace between regular expression tokens is ignored. Whitespace includes spaces, tabs and line breaks. Note that only whitespace *between* tokens is ignored. E.g. «a b c» is the same as «abc» in free-spacing mode, but «\ d» and «\d» are not the same. The former matches „ d”, while the latter matches a digit. «\d» is a single regex token composed of a backslash and a “d”. Breaking up the token with a space gives you an escaped space (which matches a space), and a literal “d”.

Likewise, grouping modifiers cannot be broken up. «(?>atomic)» is the same as «(?> ato mic)» and as «(?>ato mic)». They all match the same atomic group. They’re not the same as (? >atomic). In fact, the latter will cause a syntax error. The ?> grouping modifier is a single element in the regex syntax, and must stay together. This is true for all such constructs, including lookaround, named groups, etc.

A character class is also treated as a single token. «[abc]» is not the same as «[a b c]». The former matches one of three letters, while the latter matches those three letters or a space. In other words: free-spacing mode has no effect inside character classes. Spaces and line breaks inside character classes will be included in the character class.

This means that in free-spacing mode, you can use «\ » or «[ ]» to match a single space. Use whichever you find more readable.

Comments in Free-Spacing Mode

Another feature of free-spacing mode is that the # character starts a comment. The comment runs until the end of the line. Everything from the # until the next line break character is ignored.

Putting it all together, I could clarify the regex to match a valid date by writing it across multiple lines as:

```
# Match a 20th or 21st century date in yyyy-mm-dd format
(19|20)\d\d                # year (group 1)
[- /.]                     # separator
(0[1-9]|1[012])            # month (group 2)
[- /.]                     # separator
(0[1-9]|[12][0-9]|3[01])   # day (group 3)
```

Part 2

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 website), 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\b[^>]*>(.*?)</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]*)\b[^>]*>(.*?)</\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

Numeric Ranges. Since regular expressions work with text rather than numbers, matching specific numeric ranges requires a bit of extra care.

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.

Common Pitfalls

Catastrophic Backtracking. If your regular expression seems to take forever, or simply crashes your application, it has likely contracted a case of catastrophic backtracking. The solution is usually to be more specific about what you want to match, so the number of matches the engine has to try doesn’t rise exponentially.

Making Everything Optional. If all the parts in your regex are optional, it will match a zero-width string anywhere. Your regex will need to express the facts that different parts are optional depending on which parts are present.

Repeating a Capturing Group vs. Capturing a Repeated Group. Repeating a capturing group will capture only the last iteration of the group. Capture a repeated group if you want to capture all iterations.

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. 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}$`» .

The previous paragraph also applies to all following examples. You may need to change word boundaries into start/end-of-string anchors, or vice versa. And you will need to turn on the case insensitive matching option.

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 websites 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.

«^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.(?:[A-Z]{2}|com|org|net|gov|biz|info|name|aero|biz|info|jobs|museum)$» could be used 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 are almost 200 of them.

Email addresses can be on servers on a subdomain, e.g. „john@server.department.company.com”. All of the above regexes will match this email address, because I included a dot in the character class after the @ symbol. However, the above regexes will also match „john@aol...com” which is not valid due to the consecutive dots. You can exclude such matches by replacing «[A-Z0-9.-]+\.» with «(?:[A-Z0-9-]+\.)+» in any of the above regexes. I removed the dot from the character class and instead repeated the character class and the following literal dot. E.g. «\b[A-Z0-9._%+-]+@(?:[A-Z0-9-]+\.)+[A-Z]{2,4}\b» will match „john@server.department.company.com” but not “john@aol...com”.

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. E.g. blindly inserting this email address into a SQL will cause it to fail if strings are delimited with single quotes. 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 2822

Maybe you're wondering why there's no "official" fool-proof regex to match email addresses. Well, there is an official definition, but it's hardly fool-proof.

The official standard is known as RFC 2822. It describes the syntax that valid email addresses must adhere to. You can (but you shouldn't--read on) implement it with this regular expression:

```
«(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[\x01-
\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-
\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-
9])?|\[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-
9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-
\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])»
```

This regex has two parts: the part before the @, and the part after the @. There are two alternatives for the part before the @: it can either consist of a series of letters, digits and certain symbols, including one or more dots. However, dots may not appear consecutively or at the start or end of the email address. The other alternative requires the part before the @ to be enclosed in double quotes, allowing any string of ASCII characters between the quotes. Whitespace characters, double quotes and backslashes must be escaped with backslashes.

The part after the @ also has two alternatives. It can either be a fully qualified domain name (e.g. regular-expressions.info), or it can be a literal Internet address between square brackets. The literal Internet address can either be an IP address, or a domain-specific routing address.

The reason you shouldn't use this regex is that it only checks the basic syntax of email addresses. `john@aol.com.nospam` would be considered a valid email address according to RFC 2822. Obviously, this email address won't work, since there's no "nospam" top-level domain. It also doesn't guarantee your email software will be able to handle it. Not all applications support the syntax using double quotes or square brackets. In fact, RFC 2822 itself marks the notation using square brackets as obsolete.

We get a more practical implementation of RFC 2822 if we omit the syntax using double quotes and square brackets. It will still match 99.99% of all email addresses in actual use today.

```
«[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@(?:[a-z0-9](?:[a-
z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?»
```

A further change you could make is to allow any two-letter country code top level domain, and only specific generic top level domains. This regex filters dummy email addresses like asdf@adsf.adsf. You will need to update it as new top-level domains are added.

```
«[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@(?:[a-z0-9](?:[a-
z0-9-]*[a-z0-9])?\.)+(?:[A-
Z]{2}|com|org|net|gov|biz|info|name|aero|biz|info|jobs|museum)\b»
```

So even when following official standards, there are still trade-offs to be made. Don't blindly copy regular expressions from online libraries or discussion forums. Always test them on your own data and with your own applications.

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]*)*e».

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»

If you want to find any pair of two words out of a list of words, you can use: «\b(word1|word2|word3)(?:\W+\w+){1,6}?\W+(word1|word2|word3)\b». This regex will also find a word near itself, e.g. it will match „word2 near word2”.

9. Runaway Regular Expressions: Catastrophic Backtracking

Consider the regular expression «(x+x+)+y». Before you scream in horror and say this contrived example should be written as «(xx)+y» to match exactly the same without those terribly nested quantifiers: just assume that each "x" represents something more complex, with certain strings being matched by both "x". See the section on HTML files below for a real example.

Let's see what happens when you apply this regex to "xxxxxxxxxxy". The first «x+» will match all 10 „x" characters. The second «x+» fails. The first «x+» then backtracks to 9 matches, and the second one picks up the remaining „x". The group has now matched once. The group repeats, but fails at the first «x+». Since one repetition was sufficient, the group matches. «y» matches „y" and an overall match is found. The regex is declared functional, the code is shipped to the customer, and his computer explodes. Almost.

The above regex turns ugly when the "y" is missing from the subject string. When «y» fails, the regex engine backtracks. The group has one iteration it can backtrack into. The second «x+» matched only one „x", so it can't backtrack. But the first «x+» can give up one "x". The second «x+» promptly matches „xx". The group again has one iteration, fails the next one, and the «y» fails. Backtracking again, the second «x+» now has one backtracking position, reducing itself to match „x". The group tries a second iteration. The first «x+» matches but the second is stuck at the end of the string. Backtracking again, the first «x+» in the group's first iteration reduces itself to 7 characters. The second «x+» matches „xxx". Failing «y», the second «x+» is reduced to „xx" and then „x". Now, the group can match a second iteration, with one „x" for each «x+». But this (7,1),(1,1) combination fails too. So it goes to (6,4) and then (6,2)(1,1) and then (6,1),(2,1) and then (6,1),(1,2) and then I think you start to get the drift.

If you try this regex on a 10x string in RegexBuddy's debugger, it'll take 2559 steps to figure out the final «y» is missing. For an 11x string, it needs 5119 steps. For 12, it takes 10239 steps. Clearly we have an exponential complexity of O(2^n) here. At 16x the debugger bows out at 100,000 steps, diagnosing a bad case of catastrophic backtracking.

RegexBuddy is forgiving in that it detects it's going in circles, and aborts the match attempt. Other regex engines (like .NET) will keep going forever, while others will crash with a stack overflow (like Perl, before version 5.10). Stack overflows are particularly nasty on Windows, since they tend to make your application vanish without a trace or explanation. Be very careful if you run a web service that allows users to supply their own regular expressions. People with little regex experience have surprising skill at coming up with exponentially complex regular expressions.

Possessive Quantifiers and Atomic Grouping to The Rescue

In the above example, the sane thing to do is obviously to rewrite it as «(xx)+y» which eliminates the nested quantifiers entirely. Nested quantifiers are repeated or alternated tokens inside a group that is itself repeated or alternated. These almost always lead to catastrophic backtracking. About the only situation where they don't is when the start of each alternative inside the group is not optional, and mutually exclusive with the start of all the other alternatives, and mutually exclusive with the token that follows it (inside its alternative inside the group). E.g. «(a+b+|c+d+)+y» is safe. If anything fails, the regex engine will backtrack through the whole regex, but it will do so linearly. The reason is that all the tokens are mutually exclusive. None of them can match any characters matched by any of the others. So the match attempt at each backtracking position will fail, causing the regex engine to backtrack linearly. If you test this on "aaaabbbbccccdddd", RegexBuddy needs only 14 steps rather than 100,000+ steps to figure it out.

However, it’s not always possible or easy to rewrite your regex to make everything mutually exclusive. So we need a way to tell the regex engine not to backtrack. When we’ve grabbed all the x’s, there’s no need to backtrack. There couldn’t possibly be a “y” in anything matched by either «x+». Using a possessive quantifier, our regex becomes «(x+x+)++y». This fails the 16x string in merely 8 steps. That’s 7 steps to match all the x’s, and 1 step to figure out that «y» fails. No backtracking is done. Using an atomic group, the regex becomes «(?>(x+x+)+)y» with the exact same results.

A Real Example: Matching CSV Records

Here’s a real example from a technical support case I once handled. The customer was trying to find lines in a comma-delimited text file where the 12th item on a line started with a “P”. He was using the innocently-looking 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 possibilities 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. All this would take a long time if you ran this regex on a large CSV file where most rows don’t have a P at the start of the 12th field.

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 29,687 steps to conclude there regex cannot match "1,2,3,4,5,6,7,8,9,10,11,12". If the string is "1,2,3,4,5,6,7,8,9,10,11,12,13", just 3 characters more, the number of steps doubles to 60,315. 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 forty-eight steps to fail, whether the subject string has 12 numbers, 13 numbers, 16 numbers or a billion. While the complexity of the original regex was exponential, the complexity of the improved regex is constant with respect to whatever follows the 12th field. The reason is the regex fails immediately when it discovers the 12th field doesn't start with a P. It simply backtracks 12 times without expanding again, and that's it.

The complexity of the improved regex is linear to the length of the first 11 fields. 36 steps are needed in our example. That's the best we can do, since the engine does have to scan through all the characters of the first 11 fields to find out where the 12th one begins. Our improved regex is a perfect solution.

Alternative Solution Using Atomic Grouping

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.

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» .

Quickly Matching a Complete HTML File

Another common situation where catastrophic backtracking occurs is when trying to match "something" followed by "anything" followed by "another something" followed by "anything", where the lazy dot «.*?» is used. The more "anything", the more backtracking. Sometimes, the lazy dot is simply a symptom of a lazy programmer. «".*?"» is not appropriate to match a double-quoted string, since you don't really want to allow anything between the quotes. A string can't have (unescaped) embedded quotes, so «"[^"\r\n]*"» is more appropriate, and won't lead to catastrophic backtracking when combined in a larger regular expression. However, sometimes "anything" really is just that. The problem is that "another something" also qualifies as "anything", giving us a genuine «x+x+» situation.

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. This isn't quite as disastrous as the 2^N complexity of our first example, but will lead to very unacceptable performance on larger invalid files.

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 to 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.

10. Repeating a Capturing Group vs. Capturing a Repeated Group

When creating a regular expression that needs a capturing group to grab part of the text matched, a common mistake is to repeat the capturing group instead of capturing a repeated group. The difference is that the repeated capturing group will capture only the last iteration, while a group capturing another group that's repeated will capture all iterations. An example will make this clear. Let's say you want to match a tag like „!abc!" or „!123!". Only these two are possible, and you want to capture the „abc" or „123" to figure out which tag you got. That's easy enough: «!(abc|123)!» will do the trick.

Now let's say that the tag can contain multiple sequences of "abc" and "123", like „!abc123!" or „!123abcabc!". The quick and easy solution is «!(abc|123)+!». This regular expression will indeed match these tags. However, it no longer meets our requirement to capture the tag's label into the capturing group. When this regex matches „!abc123!", the capturing group stores only „123". When it matches „!123abcabc!", it only stores „abc".

This is easy to understand if we look at how the regex engine applies «!(abc|123)!» to "!abc123!". First, «!» matches „!". The engine then enters the capturing group. It makes note that capturing group #1 was entered when the engine reached the position between the first and second character in the subject string. The first token in the group is «abc», which matches „abc". A match is found, so the second alternative isn't tried. (The engine does store a backtracking position, but this won't be used in this example.) The engine now leaves the capturing group. It makes note that capturing group #1 was exited when the engine reached the position between the 4th and 5th characters in the string.

After having exited from the group, the engine notices the plus. The plus is greedy, so the group is tried again. The engine enters the group again, and takes note that capturing group #1 was entered between the 4th and 5th characters in the string. It also makes note that since the plus is not possessive, it may be backtracked. That is, if the group cannot be matched a second time, that's fine. In this backtracking note, the regex engine also saves the entrance and exit positions of the group during the previous iteration of the group. «abc» fails to match "123", but «123» succeeds. The group is exited again. The exit position between characters 7 and 8 is stored.

The plus allows for another iteration, so the engine tries again. Backtracking info is stored, and the new entrance position for the group is saved. But now, both «abc» and «123» fail to match "!". The group fails, and the engine backtracks. While backtracking, the engine restores the capturing positions for the group. Namely, the group was entered between characters 4 and 5, and existed between characters 7 and 8.

The engine proceeds with «!», which matches „!". An overall match is found. The overall match spans the whole subject string. The capturing group spaces characters 5, 6 and 7, or „123". Backtracking information is discarded when a match is found, so there's no way to tell after the fact that the group had a previous iteration that matched „abc". (The only exception to this is the .NET regex engine, which does preserve backtracking information for capturing groups after the match attempt.)

The solution to capturing „abc123" in this example should be obvious now: the regex engine should enter and leave the group only once. This means that the plus should be inside the capturing group rather than outside. Since we do need to group the two alternatives, we'll need to place a second capturing group around the repeated group: «!((abc|123)+)!». When this regex matches „!abc123!", capturing group #1 will store „abc123", and group #2 will store „123". Since we're not interested in the inner group's match, we can optimize this regular expression by making the inner group non-capturing: «!((?:abc|123)+)!».

Part 3

Tools & Languages

1. Specialized Tools and Utilities for Working with Regular Expressions

These tools and utilities have regular expressions as the core of their functionality.

grep - The utility from the UNIX world that first made regular expressions popular

PowerGREP - Next generation grep for Microsoft Windows

RegexBuddy - Learn, create, understand, test, use and save regular expressions. RegexBuddy makes working with regular expressions easier than ever before.

General Applications with Notable Support for Regular Expressions

There are a lot of applications these days that support regular expressions in one way or another, enhancing certain part of their functionality. But certain applications stand out from the crowd by implementing a full-featured Perl-style regular expression flavor and allowing regular expressions to be used instead of literal search terms throughout the application.

EditPad Pro - Convenient text editor with a powerful regex-based search and replace feature, as well as regex-based customizable syntax coloring.

Programming Languages and Libraries

If you are a programmer, you can save a lot of coding time by using regular expressions. With a regular expression, you can do powerful string parsing in only a handful lines of code, or maybe even just a single line. A regex is faster to write and easier to debug and maintain than dozens or hundreds of lines of code to achieve the same by hand.

Delphi - Delphi does not have built-in regex support. Delphi for .NET can use the .NET framework regex support. For Win32, there are several PCRE-based VCL components available.

Java - Java 4 and later include an excellent regular expressions library in the java.util.regex package.

JavaScript - If you use JavaScript to validate user input on a web page at the client side, using JavaScript’s built-in regular expression support will greatly reduce the amount of code you need to write.

.NET (dot net) - Microsoft’s new development framework includes a poorly documented, but very powerful regular expression package, that you can use in any .NET-based programming language such as C# (C sharp) or VB.NET.

PCRE - Popular open source regular expression library written in ANSI C that you can link directly into your C and C++ applications, or use through an .so (UNIX/Linux) or a .dll (Windows).

Perl - The text-processing language that gave regular expressions a second life, and introduced many new features. Regular expressions are an essential part of Perl.

PHP - Popular language for creating dynamic web pages, with three sets of regex functions. Two implement POSIX ERE, while the third is based on PCRE.

POSIX - The POSIX standard defines two regular expression flavors that are implemented in many applications, programming languages and systems.

Python - Popular high-level scripting language with a comprehensive built-in regular expression library

REALbasic - Cross-platform development tool similar to Visual Basic, with a built-in RegEx class based on PCRE.

Ruby - Another popular high-level scripting language with comprehensive regular expression support as a language feature.

Tcl - Tcl, a popular “glue” language, offers three regex flavors. Two POSIX-compatible flavors, and an “advanced” Perl-style flavor.

VBScript - Microsoft scripting language used in ASP (Active Server Pages) and Windows scripting, with a built-in RegExp object implementing the regex flavor defined in the JavaScript standard.

Visual Basic 6 - Last version of Visual Basic for Win32 development. You can use the VBScript RegExp object in your VB6 applications.

XML Schema - The W3C XML Schema standard defines its own regular expression flavor for validating simple types using pattern facets.

Databases

Modern databases often offer built-in regular expression features that can be used in SQL statements to filter columns using a regular expression. With some databases you can also use regular expressions to extract the useful part of a column, or to modify columns using a search-and-replace.

MySQL - MySQL’s REGEXP operator works just like the LIKE operator, except that it uses a POSIX Extended Regular Expression.

Oracle - Oracle Database 10g adds 4 regular expression functions that can be used in SQL and PL/SQL statements to filter rows and to extract and replace regex matches. Oracle implements POSIX Extended Regular Expressions.

PostgreSQL - PostgreSQL provides matching operators and extraction and substitution functions using the “Advanced Regular Expression” engine also used by Tcl.

2. 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.

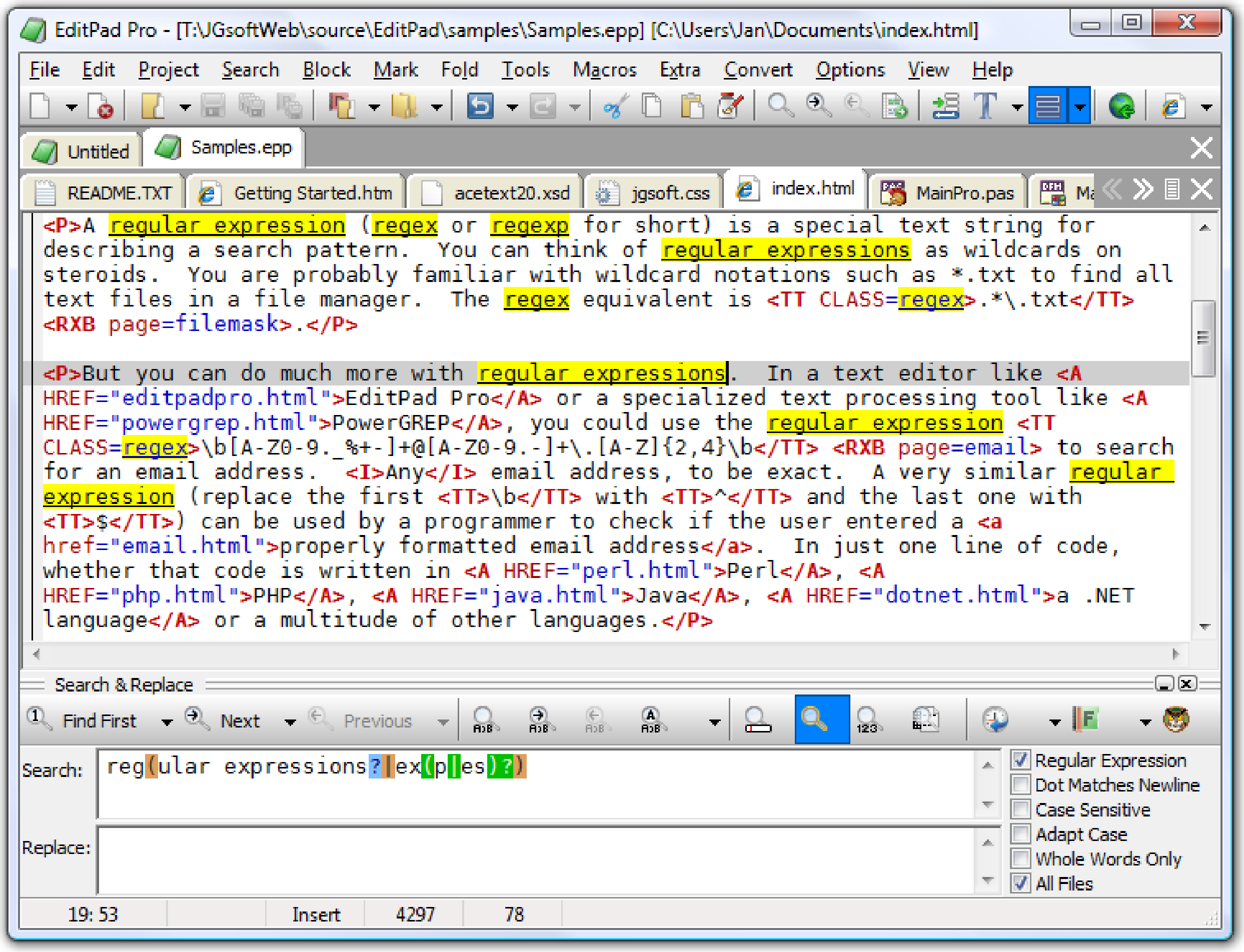
There are many PCRE-based VCL components available. Most are free, some are not. Some compile PCRE into a DLL that you need to ship along with your application, others link the PCRE OBJ files directly into your Delphi EXE.

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.

3. EditPad Pro: Convenient Text Editor with Full Regular Expression Support

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.



```
<P>A regular expression (regex or regexp for short) is a special text string for
describing a search pattern.  You can think of regular expressions as wildcards on
steroids.  You are probably familiar with wildcard notations such as *.txt to find all
text files in a file manager.  The regex equivalent is <TT CLASS=regex>.*\.txt</TT>
<RXB page=filemask>.</P>

<P>But you can do much more with regular expressions.  In a text editor like <A
HREF="editpadpro.html">EditPad Pro</A> or a specialized text processing tool like <A
HREF="powergrep.html">PowerGREP</A>, you could use the regular expression <TT
CLASS=regex>\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b</TT> <RXB page=email> to search
for an email address.  <I>Any</I> email address, to be exact.  A very similar regular
expression (replace the first <TT>\b</TT> with <TT>^</TT> and the last one with
<TT>$</TT>) can be used by a programmer to check if the user entered a <a
href="email.html">properly formatted email address</a>.  In just one line of code,
whether that code is written in <A HREF="perl.html">Perl</A>, <A
HREF="php.html">PHP</A>, <A HREF="java.html">Java</A>, <A HREF="dotnet.html">a .NET
language</A> or a multitude of other languages.</P>
```

Search: `reg(ular expressions?|ex(p|es)?)`

Replace:

☑ Regular Expression
□ Dot Matches Newline
□ Case Sensitive
□ Adapt Case
□ Whole Words Only
☑ All Files

19: 53 | Insert | 4297 | 78

EditPad Pro’s Regular Expression Support

EditPad Pro doesn’t use a limited and outdated regular expression engine like so many other text editors do. EditPad Pro uses the same full-featured regular expression engine used by PowerGREP and RegexBuddy. EditPad Pro’s regex flavor is fully compatible with the flavors used by Perl, Java, .NET and many other modern Perl-style regular expression flavors. 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, and hit the Find First or Find Next button. EditPad Pro will then highlight search match. If the search pane takes up too much space, simply close it after entering the regular expression. Press Ctrl+F3 to find the first match, or F3 to find the next one.

When there are no further regex matches, EditPad Pro doesn't interrupt you with a popup message that you have to OK. The text cursor and selection will simply stay where they were, and the find button that you clicked will flash briefly. This may seem a little subtle at first, but you'll quickly appreciate EditPad Pro staying out of your way and keeping you productive.

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 the Replace Next button. Or, click Replace All to get it over with.

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. Syntax coloring schemes can be shared on the EditPad Pro website.

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.

File Navigation Schemes for Text Folding and Navigation

Text editors catering to programmers often allow you to fold certain sections in source code files to get a better overview. Another common feature is a sidebar showing you the file's structure, enabling you to quickly jump to a particular class definition or method implementation.

EditPad Pro also offers both these features, with one key difference. Most text editors only support folding and navigation for a limited set of file types, usually the more popular programming languages. If you use a less common language or file format, not to mention a custom one, you're out of luck.

EditPad Pro, however, implements folding and navigation using file navigation schemes. A bunch of them are included with EditPad Pro. These schemes are fully editable, and you can even create your own. Many file navigation schemes have been shared by other EditPad Pro users.

You can create and edit this schemes with a special file navigation scheme editor, which you can download after buying EditPad Pro. Like the syntax coloring schemes, file navigation schemes are based entirely on regular expressions. Because file navigation schemes are extremely flexible, editing them will take some effort. But with a bit of practice, you can make EditPad Pro's code folding and file navigation to work just the way you want it, and support all the file types that you work with, even proprietary ones.

More Information on EditPad Pro and Free Trial Download

EditPad Pro works under Windows 98, ME, NT4, 2000, XP and Vista. For more information on EditPad Pro, please visit www.editpadpro.com.

4. 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.

5. Using Regular Expressions in Java

Java 4 (JDK 1.4) 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.

Java 5 and 6 use the same regular expression flavor (with a few minor fixes), and provide the same regular expression classes. They add a few advanced functions not discussed on this page.

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.

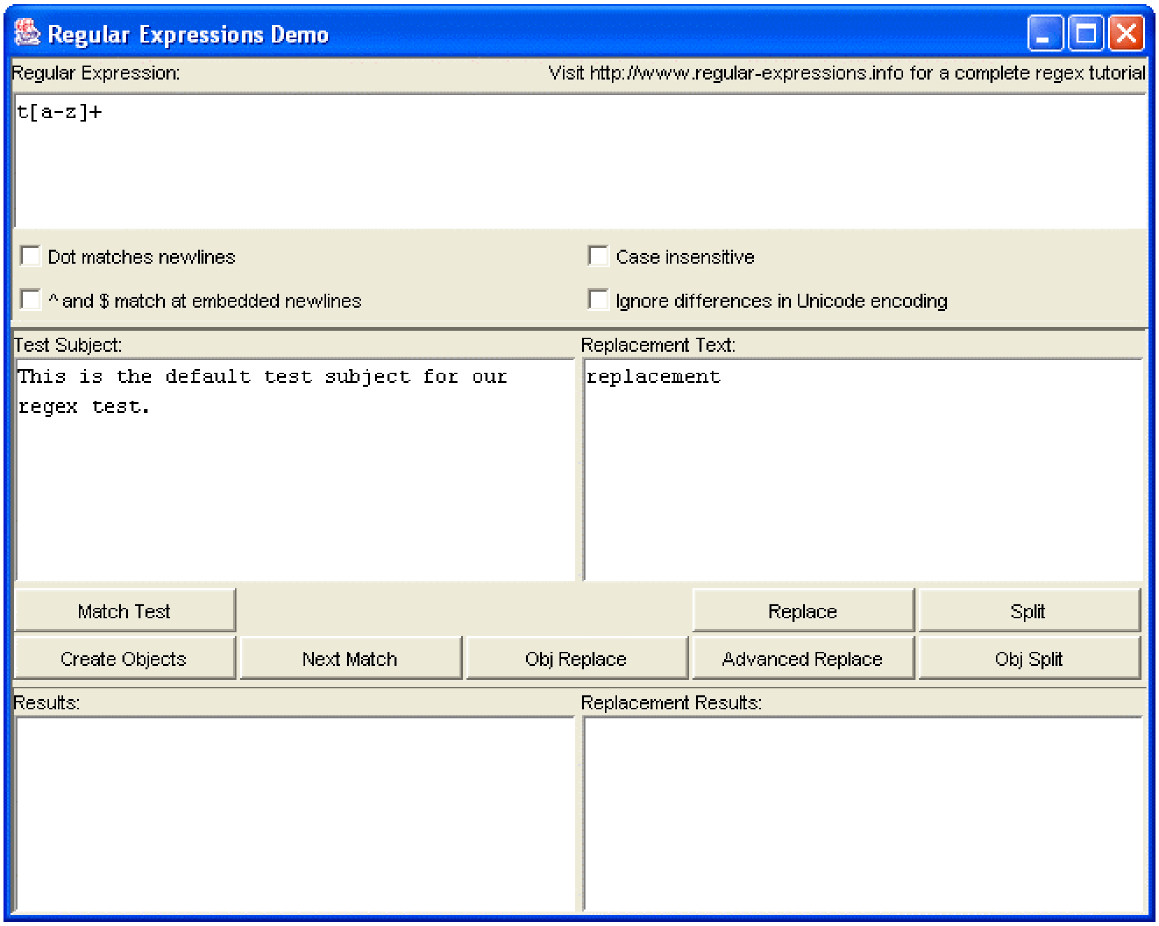
Java Demo Application using Regular Expressions

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
```



Regular Expression:

Visit http://www.regular-expressions.info for a complete regex tutorial

```
t[a-z]+
```

□ Dot matches newlines

□ Case insensitive

□ ^ and $ match at embedded newlines

□ Ignore differences in Unicode encoding

Test Subject:

```
This is the default test subject for our
regex test.
```

Replacement Text:

```
replacement
```

Results:

Replacement Results:

```
/** 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 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. JavaScript's regular expression flavor is part of the ECMA-262 standard for the language. This means your regular expressions should work exactly the same in all implementations of JavaScript (i.e. in different web browsers).

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.

JavaScript implements Perl-style regular expressions. However, it lacks quite a number of advanced features available in Perl and other modern regular expression flavors:

- 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, except for matching single characters with `\uFFFF`
- No named capturing groups. Use numbered capturing groups instead.
- No mode modifiers to set matching options within the regular expression.
- No conditionals.
- No regular expression comments. Describe your regular expression with JavaScript `//` comments instead, outside the regular expression string.

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. JavaScript RegExp Example: Regular Expression Tester

```
<SCRIPT LANGUAGE=“JavaScript”><!--
function demoMatchClick() {
  var re = new RegExp(document.demoMatch.regex.value);
  if (document.demoMatch.subject.value.match(re)) {
    alert(“Successful match”);
  } else {
    alert(“No match”);
  }
}

function demoShowMatchClick() {
  var re = new RegExp(document.demoMatch.regex.value);
  var m = re.exec(document.demoMatch.subject.value);
  if (m == null) {
    alert(“No match”);
  } else {
    var s = “Match at position ” + m.index + ":\n";
    for (i = 0; i < m.length; i++) {
      s = s + m[i] + "\n";
    }
    alert(s);
  }
}

function demoReplaceClick() {
  var re = new RegExp(document.demoMatch.regex.value, “g”);
  document.demoMatch.result.value =
    document.demoMatch.subject.value.replace(re,
      document.demoMatch.replacement.value);
}
// -->
</SCRIPT>

<FORM ID=“demoMatch” NAME=“demoMatch” METHOD=POST ACTION="javascript:void(0)">
<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 JavaScript RegExp object” SIZE=50></P>
<P><INPUT TYPE=SUBMIT VALUE=“Test Match” ONCLICK="demoMatchClick()">
<INPUT TYPE=SUBMIT VALUE=“Show Match” ONCLICK="demoShowMatchClick()"></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=SUBMIT VALUE=“Replace” ONCLICK="demoReplaceClick()"></P>
</FORM>
```

9. MySQL Regular Expressions with The REGEXP Operator

MySQL's support for regular expressions is rather limited, but still very useful. MySQL only has one operator that allows you to work with regular expressions. This is the REGEXP operator, which works just like the LIKE operator, except that instead of using the `_` and `%` wildcards, it uses a POSIX Extended Regular Expression (ERE). Despite the "extended" in the name of the standard, the POSIX ERE flavor is a fairly basic regex flavor by modern standards, as you can see in the regex flavor comparison in this book. Still, it makes the REGEXP operator far more powerful and flexible than the simple LIKE operator.

One important difference between the LIKE and REGEXP operators is that the LIKE operator only returns True if the pattern matches the whole string. E.g. `WHERE testcolumn LIKE 'jg'` will return only rows where testcolumn is identical to "`jg`", except for differences in case perhaps. On the other hand, `WHERE testcolumn REGEXP 'jg'` will return all rows where testcolumn has "`jg`" anywhere in the string. Use `WHERE testcolumn REGEXP '^jg$'` to get only columns identical to "`jg`". The equivalent of `WHERE testcolumn LIKE 'jg%'` would be `WHERE testcolumn REGEXP '^jg'`. There's no need to put a «`.*`» at the end of the regex (the REGEXP equivalent of LIKE's `%`), since partial matches are accepted.

MySQL does not offer any matching modes. POSIX EREs don't support mode modifiers inside the regular expression, and MySQL's REGEXP operator does not provide a way to specify modes outside the regular expression. The REGEXP operator always applies regular expressions case insensitively, the dot matches all characters including newlines, and the caret and dollar only match at the very start and end of the string. In other words: MySQL treats newline characters like ordinary characters.

Remember that MySQL supports C-style escape sequences in strings. While POSIX ERE does not support tokens like «`\n`» to match non-printable characters like line breaks, MySQL does support this escape in its strings. So "`WHERE testcolumn REGEXP '\n'`" returns all rows where testcolumn contains a line break. MySQL converts the `\n` in the string into a single line break character before parsing the regular expression. This also means that backslashes need to be escaped. The regex «`\\`» to match a single backslash becomes `'\\\\'` as a MySQL string, and the regex «`\$`» to match a dollar symbol becomes `'\\$'` as a MySQL string. All this is unlike other databases like Oracle, which don't support `\n` and don't require backslashes to be escaped.

To return rows where the column doesn't match the regular expression, use `WHERE testcolumn NOT REGEXP 'pattern'` The RLIKE operator is a synonym of the REGEXP operator. `WHERE testcolumn RLIKE 'pattern'` and `WHERE testcolumn NOT RLIKE 'pattern'` are identical to `WHERE testcolumn REGEXP 'pattern'` and `WHERE testcolumn NOT REGEXP 'pattern'`. I recommend you use REGEXP instead of RLIKE, to avoid confusion with the LIKE operator.

10. 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 recommend 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.

As you can see in the regular expression flavor comparison, .NET's regex flavor is very feature-rich. The only noteworthy feature that's lacking are possessive quantifiers.

There are no differences in the regex flavor supported by .NET versions 1.x, 2.0 and 3.0, except for one feature added in .NET 2.0: character class subtraction. It works exactly the way it does in XML Schema regular expressions. The XML Schema standard first defined this feature and its syntax.

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"`.

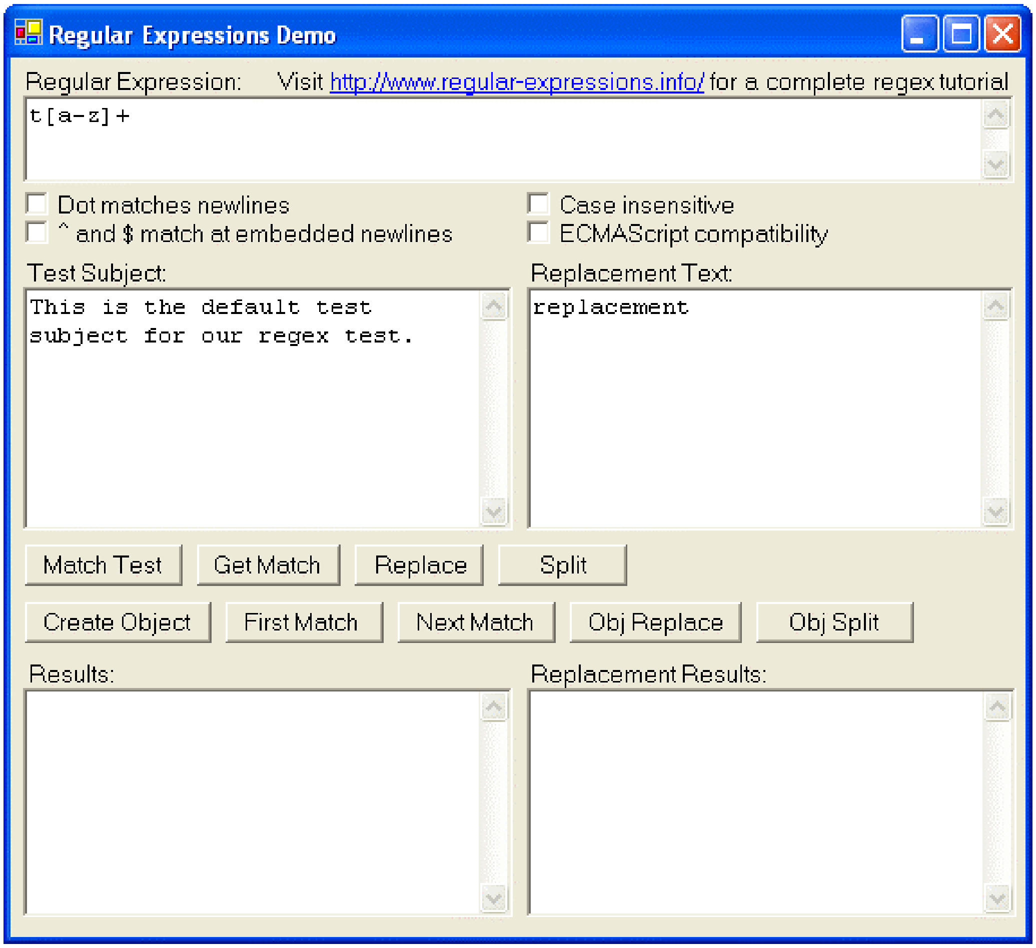
.NET Framework Demo Application using Regular Expressions (C# Syntax)

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.

11. 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
```



Regular Expression: Visit http://www.regular-expressions.info/ for a complete regex tutorial

```
t[a-z]+
```

□ Dot matches newlines
□ ^ and $ match at embedded newlines
□ Case insensitive
□ ECMAScript compatibility

Test Subject:

```
This is the default test
subject for our regex test.
```

Replacement Text:

```
replacement
```

Results:

Replacement Results:

```
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*/ );
  }
}

}
}
```

12. Oracle Database 10g Regular Expressions

With version 10g, Oracle Database offers 4 regexp functions that you can use in SQL and PL/SQL statements. These functions implement the POSIX Extended Regular Expressions (ERE) standard. Oracle fully supports collating sequences and equivalence classes in bracket expressions. The NLS_SORT setting determines the POSIX locale used, which determines the available collating sequences and equivalence classes.

Oracle does not implement the POSIX ERE standard exactly, however. It deviates in three areas. First, Oracle supports the backreferences `\1` through `\9` in the regular expression. The POSIX ERE standard does not support these, even though POSIX BRE does. In a fully compliant engine, `\1` through `\9` would be illegal. The POSIX standard states it is illegal to escape a character that is not a metacharacter with a backslash. Oracle allows this, and simply ignores the backslash. E.g. «`\z`» is identical to «`z`» in Oracle. The result is that all POSIX ERE regular expressions can be used with Oracle, but some regular expressions that work in Oracle may cause an error in a fully POSIX-compliant engine. Obviously, if you only work with Oracle, these differences are irrelevant.

The third difference is more subtle. It won’t cause any errors, but may result in different matches. As I explained in the topic about the POSIX standard, it requires the regex engine to return the longest match in case of alternation. Oracle’s engine does not do this. It is a traditional NFA engine, like all non-POSIX regex flavors discussed in this book.

If you’ve worked with regular expressions in other programming languages, be aware that POSIX does not support non-printable character escapes like `\t` for a tab or `\n` for a newline. You can use these with a POSIX engine in a programming language like C++, because the C++ compiler will interpret the `\t` and `\n` in string constants. In SQL statements, you’ll need to type an actual tab or line break in the string with your regular expression to make it match a tab or line break. Oracle’s regex engine will interpret the string `'\t'` as the regex «`t`» when passed as the regexp parameter.

Oracle’s Four REGEXP Functions

Oracle Database 10g offers four regular expression functions. You can use these equally in your SQL and PL/SQL statements.

`REGEXP_LIKE(source, regexp, modes)` is probably the one you’ll use most. You can use it in the WHERE and HAVING clauses of a SELECT statement. In a PL/SQL script, it returns a Boolean value. You can also use it in a CHECK constraint. The `source` parameter is the string or column the regex should be matched against. The `regexp` parameter is a string with your regular expression. The `modes` parameter is optional. It sets the matching modes.

```
SELECT * FROM mytable WHERE REGEXP_LIKE(mycolumn, ’regexp’, ’i’);
IF REGEXP_LIKE(’subject’, ’regexp’) THEN /* Match */ ELSE /* No match */ END IF;
ALTER TABLE mytable ADD (CONSTRAINT mycolumn_regexp CHECK (REGEXP_LIKE(mycolumn, ’^regexp$’)));
```

`REGEXP_SUBSTR(source, regexp, position, occurrence, modes)` returns a string with the part of `source` matched by the regular expression. If the match attempt fails, NULL is returned. You can use REGEXP_SUBSTR with a single string or with a column. You can use it in SELECT clauses to retrieve only a certain part of a column. The `position` parameter specifies the character position in the source string at which the match attempt should start. The first character has position 1. The `occurrence` parameter specifies which match to get. Set it to 1 to get the first match. If you specify a higher number, Oracle will continue to attempt to match the regex starting at the end of the previous match, until it found as many matches as you specified. The last match is then returned. If there are fewer matches, NULL is returned. Do not confuse this parameter with backreferences. Oracle does not provide a function to return the part of the string matched by a capturing group. The last three parameters are optional.

```
SELECT REGEXP_SUBSTR(mycolumn, ’regexp’) FROM mytable;
match := REGEXP_SUBSTR(’subject’, ’regexp’, 1, 1, ’i’)
```

`REGEXP_REPLACE``(source, regexp, replacement, position, occurrence, modes)` returns the source string with one or all regex matches replaced. If no matches can be found, the original string is replaced. If you specify a positive number for `occurrence` (see the above paragraph) only that match is replaced. If you specify zero or omit the parameter, all matches are replaced. The last three parameters are optional. The `replacement` parameter is a string that each regex match will be replaced with. You can use the backreferences `\1` through `\9` in the replacement text to re-insert text matched by a capturing group. You can reference the same group more than once. There’s no replacement text token to re-insert the whole regex match. To do that, put parentheses around the whole regexp, and use `\1` in the replacement. If you want to insert `\1` literally, use the string `'\\1'`. Backslashes only need to be escaped if they’re followed by a digit or another backslash. To insert `\\` literally, use the string `'\\\\'`. While SQL does not require backslashes to be escaped in strings, the REGEXP_REPLACE function does.

```
SELECT REGEXP_REPLACE(mycolumn, ’regexp’, ’replacement’) FROM mytable;
result := REGEXP_REPLACE(’subject’, ’regexp’, ’replacement’, 1, 0, ’i’);
```

`REGEXP_INSTR``(source, regexp, position, occurrence, return_option, modes)` returns the beginning or ending position of a regex match in the source string. This function takes the same parameters as REGEXP_SUBSTR, plus one more. Set `return_option` to zero or omit the parameter to get the position of the first character in match. Set it to one to get the position of the first character after the match. The first character in the string has position 1. REGEXP_INSTR returns zero if the match cannot be found. The last 4 parameters are optional.

```
SELECT REGEXP_INSTR(mycolumn, ’regexp’, 1, 1, 0, ’i’) FROM mytable;
```

Oracle’s Matching Modes

The `modes` parameter that each of the four regexp functions accepts should be a string of up to three characters, out of four possible characters. E.g. `'i'` turns on case insensitive matching, while `'inm'` turns on those three options. `'i'` and `'c'` are mutually exclusive. If you omit this parameter or pass an empty string, the default matching modes are used.

- `'i'`: Turn on case insensitive matching. The default depends on the NLS_SORT setting.
- `'c'`: Turn on case sensitive matching. The default depends on the NLS_SORT setting.
- `'n'`: Make the dot match any character, including newlines. By default, the dot matches any character except newlines.
- `'m'`: Make the caret and dollar match at the start and end of each line (i.e. after and before line breaks embedded in the source string). By default, these only match at the very start and the very end of the string.

13. 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. The PHP preg functions and the REALbasic RegEx class are built on top of PCRE.

PCRE implements almost the entire Perl 5.8 regular expression syntax. Only the support for various Unicode properties with `\p` is incomplete, though the most important ones are supported.

Using PCRE 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/.

Compiling PCRE with Unicode Support

By default, PCRE compiles without Unicode support. If you try to use `\p`, `\P` or `\X` in your regular expressions, PCRE will complain it was compiled without Unicode support.

To compile PCRE with Unicode support, you need to define the SUPPORT_UTF8 and SUPPORT_UCP conditional defines. If PCRE's configuration script works on your system, you can easily do this by running `./configure --enable-unicode-properties` before running `make`.

14. 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 website. 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;`.

15. PHP Provides Three Sets of Regular Expression Functions

PHP is an open source language for producing dynamic web pages, similar to ASP. PHP has three sets of functions that allow you to work with regular expressions. Each set has its advantages and disadvantages.

The first set of regex functions are those that start with `ereg`. They implement POSIX Extended Regular Expressions, like the traditional UNIX egrep command. The advantage of the `ereg` functions is that they are supported by all versions of PHP 3, 4 and 5, and are part of the PHP codebase itself. However, many of the more modern regex features such as lazy quantifiers, lookaround and Unicode are not supported by the `ereg` functions. Don't let the "extended" moniker fool you. The POSIX standard was defined in 1986, and regular expressions have come a long way since then.

The second set is a variant of the first, prefixing `mb_` for "multibyte" to the function names. While `ereg` treats the regex and subject string as a series of 8-bit characters, `mb_ereg` can work with multi-byte characters from various code pages. If you want your regex to treat Far East characters as individual characters, you'll either need to use the `mb_ereg` functions, or the `preg` functions with the `/u` modifier. `mb_ereg` is available in PHP 4.2.0 and later. It uses the same POSIX ERE flavor.

The third 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 `preg` offer a far richer regex flavor, a more complete function set, and are often faster too.

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.

`int` **`ereg`** `(string pattern, string subject [, array groups])` returns the length of the match if the regular expression pattern matches the subject string or part of the subject string, or zero otherwise. Since zero evaluates to False and non-zero evaluates to True, you can use `ereg` in an `if` statement to test for a match. 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 mb_ereg Function Set

The mb_ereg functions work exactly the same as the ereg functions, with one key difference: while ereg treats the regex and subject string as a series of 8-bit characters, mb_ereg can work with multi-byte characters from various code pages. E.g. encoded with Windows code page 936 (Simplified Chinese), the word “中国” (“China”) consists of four bytes: D6D0B9FA. Using the ereg function with the regular expression «.» on this string would yield the first byte D6 as the result. The dot matched exactly one byte, as the ereg functions are byte-oriented. Using the mb_ereg function after calling mb_regex_encoding("CP936") would yield the bytes D6D0 or the first character “中” as the result.

To make sure your regular expression uses the correct code page, call mb_regex_encoding() to set the code page. If you don’t, the code page returned by or set by mb_internal_encoding() is used instead.

If your PHP script uses UTF-8, you can use the preg functions with the /u modifier to match multi-byte UTF-8 characters instead of individual bytes. The preg functions do not support any other code pages.

See the PHP manual for more information on the mb_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\//'. Just like Perl, the preg functions allow any non-alphanumeric character as regex delimiters. The URL regex would be more readable as '%http://www\.jgsoft\.com/%' using percentage signs as the regex delimiters.

Unlike programming languages like C# or Java, PHP does not require all backslashes in strings to be escaped. If you want to include a backslash as a literal character in a PHP string, you only need to escape it if it is followed by another character that needs to be escaped. In single quoted-strings, only the single quote and the backslash itself need to be escaped. That is why in the above regex, I didn’t have to double the backslashes in front of the literal dots. The regex «\\» to match a single backslash would become '/\\\\/' as a PHP preg string. Unless you want to use variable interpolation in your regular expression, you should always use single-quoted strings for regular expressions in PHP, to avoid messy duplication of backslashes.

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. '/regex/x' turns on free-spacing mode. You can specify multiple letters to turn on several options. '/regex/misx' turns on all four options.

A special option is the /u which turns on the Unicode matching mode, instead of the default 8-bit matching mode. You should specify /u for regular expressions that use «\x{FFFF}», «\X» or «\p{L}» to match Unicode characters, graphemes, properties or scripts. PHP will interpret '/regex/u' as a UTF-8 string rather than as an ASCII string.

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

16. POSIX Basic Regular Expressions

POSIX or “Portable Operating System Interface for uniX” is a collection of standards that define some of the functionality that a (UNIX) operating system should support. One of these standards defines two flavors of regular expressions. Commands involving regular expressions, such as grep and egrep, implement these flavors on POSIX-compliant UNIX systems. Several database systems also use POSIX regular expressions.

The Basic Regular Expressions or BRE flavor standardizes a flavor similar to the one used by the traditional UNIX grep command. This is pretty much the oldest regular expression flavor still in use today. One thing that sets this flavor apart is that most metacharacters require a backslash to give the metacharacter its flavor. Most other flavors, including POSIX ERE, use a backslash to suppress the meaning of metacharacters. Using a backslash to escape a character that is never a metacharacter is an error.

A BRE supports POSIX bracket expressions, which are similar to character classes in other regex flavors, with a few special features. Shorthands are not supported. Other features using the usual metacharacters are the dot to match any character except a line break, the caret and dollar to match the start and end of the string, and the star to repeat the token zero or more times. To match any of these characters literally, escape them with a backslash.

The other BRE metacharacters require a backslash to give them their special meaning. The reason is that the oldest versions of UNIX grep did not support these. The developers of grep wanted to keep it compatible with existing regular expressions, which may use these characters as literal characters. The BRE «`a{1,2}`» matches „`a{1,2}`” literally, while «`a\{1,2\}`» matches „`a`” or „`aa`”. Some implementations support `\?` and `\+` as an alternative syntax to `\{0,1\}` and `\{1,\}`, but `\?` and `\+` are not part of the POSIX standard. Tokens can be grouped with `\(` and `\)`. Backreferences are the usual `\1` through `\9`. Only up to 9 groups are permitted. E.g. «`\(ab\)\1`» matches „`abab`”, while `(ab)\1` is invalid since there’s no capturing group corresponding to the backreference `\1`. Use «`\\1`» to match „`\1`” literally.

POSIX BRE does not support any other features. Even alternation is not supported.

POSIX Extended Regular Expressions

The Extended Regular Expressions or ERE flavor standardizes a flavor similar to the one used by the UNIX egrep command. “Extended” is relative to the original UNIX grep, which only had bracket expressions, dot, caret, dollar and star. An ERE support these just like a BRE. Most modern regex flavors are extensions of the ERE flavor. By today’s standard, the POSIX ERE flavor is rather bare bones. The POSIX standard was defined in 1986, and regular expressions have come a long way since then, as you can see in the regex flavor comparison in this book.

The developers of egrep did not try to maintain compatibility with grep, creating a separate tool instead. Thus egrep, and POSIX ERE, add additional metacharacters without backslashes. You can use backslashes to suppress the meaning of all metacharacters, just like in modern regex flavors. Escaping a character that is not a metacharacter is an error.

The quantifiers `?`, `+`, `{n}`, `{n,m}` and `{n,}` repeat the preceding token zero or once, once or more, n times, between n and m times, and n or more times, respectively. Alternation is supported through the usual vertical bar `|`. Unadorned parentheses create a group, e.g. «`(abc){2}`» matches „`abcabc`”. The POSIX standard does not define backreferences. Some implementations do support \1 through \9, but these are not part of the standard for ERE. ERE is an extension of the old UNIX grep, not of POSIX BRE.

And that's exactly how far the extension goes.

POSIX ERE Alternation Returns The Longest Match

In the tutorial topic about alternation, I explained that the regex engine will stop as soon as it finds a matching alternative. The POSIX standard, however, mandates that the longest match be returned. When applying «Set|SetValue» to "SetValue", a POSIX-compliant regex engine will match „SetValue" entirely. Even if the engine is a regex-directed NFA engine, POSIX requires that it simulates DFA text-directed matching by trying all alternatives, and returning the longest match, in this case „SetValue". A traditional NFA engine would match „Set", as do all other regex flavors discussed in this book.

A POSIX-compliant engine will still find the leftmost match. If you apply «Set|SetValue» to "Set or SetValue" once, it will match „Set". The first position in the string is the leftmost position where our regex can find a valid match. The fact that a longer match can be found further in the string is irrelevant. If you apply the regex a second time, continuing at the first space in the string, then „SetValue" will be matched. A traditional NFA engine would match „Set" at the start of the string as the first match, and „Set" at the start of the 3rd word in the string as the second match.

17. PostgreSQL Has Three Regular Expression Flavors

PostgreSQL 7.4 and later use the exact same regular expression engine that was developed by Henry Spencer for Tcl 8.2. This means that PostgreSQL supports the same three regular expressions flavors: Tcl Advanced Regular Expressions, POSIX Extended Regular Expressions and POSIX Basic Regular Expressions. Just like in Tcl, AREs are the default. All my comments on Tcl's regular expression flavor, like the unusual mode modifiers and word boundary tokens, fully apply to PostgreSQL. You should definitely review them if you're not familiar with Tcl's AREs. PostgreSQL's regexp_replace function does not use the same syntax for the replacement text as Tcl's regsub command, however.

PostgreSQL versions prior to 7.4 supported POSIX Extended Regular Expressions only. If you are migrating old database code to a new version of PostgreSQL, you can set PostgreSQL's `"regex_flavor“` run-time parameter to `”extended“` instead of the default `”advanced"` to make EREs the default.

PostgreSQL also supports the traditional SQL LIKE operator, and the SQL:1999 SIMILAR TO operator. These use their own pattern languages, which are not discussed here. AREs are far more powerful, and no more complicated if you don't use functionality not offered by LIKE or SIMILAR TO.

The Tilde Operator

The tilde infix operator returns true or false depending on whether a regular expression can match part of a string, or not. E.g. `'subject' ~ 'regexp'` returns false, while `'subject' ~ '\\w'` returns true. If the regex must match the whole string, you'll need to use anchors. E.g. `'subject' ~ '^\\w$'` returns false, while `'subject' ~ '^\\w+$'` returns true. There are 4 variations of this operator:

- `~` attempts a case sensitive match
- `~*` attempts a case insensitive match
- `!~` attempts a case sensitive match, and returns true if the regex does not match any part of the subject string
- `!~*` attempts a case insensitive match, and returns true if the regex does not match any part of the subject string

While only case sensitivity can be toggled by the operator, all other options can be set using mode modifiers at the start of the regular expression. Mode modifiers override the operator type. E.g. '(?c)regex' forces the to be regex case sensitive.

The most common use of this operator is to select rows based on whether a column matches a regular expression, e.g.:

```
select * from mytable where mycolumn ~* 'regexp'
```

Regular Expressions as Literal PostgreSQL Strings

The backslash is used to escape characters in PostgreSQL strings. So a regular expression like «`\w`» that contains a backslash becomes `'\\w'` when written as a literal string in a PostgreSQL statement. To match a single literal backslash, you'll need the regex «`\\`» which becomes `'\\\\'` in PostgreSQL.

PostgreSQL Regexp Functions

With the `substring(string from pattern)` function, you can extract part of a string or column. It takes two parameters: the string you want to extract the text from, and the pattern the extracted text should match. If there is no match, `substring()` returns null. E.g. `substring('subject' from 'regexp')` returns null. If there is a match, and the regex has one or more capturing groups, the text matched by the first capturing group is returned. E.g. `substring('subject' from 's(\\w)')` returns ’u’. If there is a match, but the regex has no capturing groups, the whole regex match is returned. E.g. `substring('subject' from 's\\w')` returns ’su’. If the regex matches the string more than once, only the first match is returned. Since the `substring()` function doesn’t take a “flags” parameter, you’ll need to toggle any matching options using mode modifiers.

This function is particularly useful to extract information from columns. E.g. to extract the first number from the column *mycolumn* for each row, use:

```
select substring(mycolumn from '\d+') from mytable
```

With `regexp_replace(subject, pattern, replacement [, flags])` you can replace regex matches in a string. If you omit the flags parameter, the regex is applied case sensitively, and only the first match is replaced. If you set the flags to `'i'`, the regex is applied case insensitively. The `'g'` flag (for “global”) causes all regex matches in the string to be replaced. You can combine both flags as `'gi'`.

You can use the backreferences `\1` through `\9` in the replacement text to re-insert the text matched by a capturing group into the regular expression. `\&` re-inserts the whole regex match. Remember to double up the backslashes in literal strings.

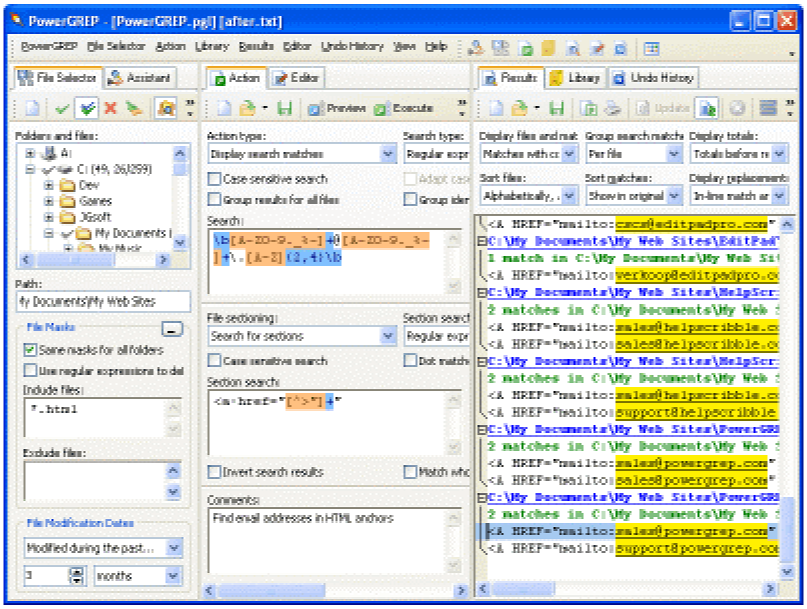
E.g. `regexp_replace('subject', '(\w)\w', '\&\1', 'g')` returns `'susbjbecet'`.

18. 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.



Folders and files:

Path: My Documents\My Web Sites

File Masks

☑ Same masks for all folders

□ Use regular expressions to def

Include files: `*.html`

Exclude files:

File Modification Dates

Modified during the past... 3 months

Action type: Display search matches

Search type: Regular expr

□ Case sensitive search

□ Adapt case

□ Group results for all files

□ Group ider

Search:

```
\b[A-Z0-9._%-]+@[A-Z0-9._%-
]+\.[A-Z]{2,4}\b
```

File sectioning: Search for sections

Section search: Regular expr

□ Case sensitive search

□ Dot match

Section search:

```
<a href="[^>"]+"
```

□ Invert search results

□ Match whc

Comments: Find email addresses in HTML anchors

Display files and mat: Matches with c

Group search matche: Per file

Display totals: Totals before r

Sort files: Alphabetically, .

Sort matches: Show in original

Display replacement: In-line match ar

```
<A HREF="mailto:cscs@editpadpro.com"
C:\My Documents\My Web Sites\EditPad
1 match in C:\My Documents\My Web Si
<A HREF="mailto:verkoop@editpadpro.co
C:\My Documents\My Web Sites\HelpScr
2 matches in C:\My Documents\My Web
<A HREF="mailto:sales@helpscribble.co
<A HREF="mailto:sales@helpscribble.co
C:\My Documents\My Web Sites\HelpScr
2 matches in C:\My Documents\My Web
<A HREF="mailto:sales@helpscribble.co
<A HREF="mailto:support@helpscribble
C:\My Documents\My Web Sites\PowerGR
2 matches in C:\My Documents\My Web
<A HREF="mailto:sales@powergrep.com"
<A HREF="mailto:sales@powergrep.com"
C:\My Documents\My Web Sites\PowerGR
2 matches in C:\My Documents\My Web
<A HREF="mailto:sales@powergrep.com"
<A HREF="mailto:support@powergrep.co
```

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, XP and Vista. For more information on PowerGREP, please visit www.powergrep.com.

19. 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 significant features missing from Python’s regex syntax are atomic grouping, possessive quantifiers and Unicode properties.

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.

Unicode

Python’s re module does not support any Unicode regular expression tokens. However, Python Unicode strings do support the `\uFFFF` notation, and Python’s re module can use Unicode strings. So you could pass the Unicode string `u"\u00E0\\d"` to the re module to match „à” followed by a digit. Note that the backslash for «`\d`» was escaped, while the one for `\u` was not. That’s because «`\d`» is a regular expression token, and a regular expression backslash needs to be escaped. `\u00E0` is a Python string token that shouldn’t be escaped. The string `u"\u00E0\\d"` is seen by the regular expression engine as «`à\d`».

If you did put another backslash in front of the `\u`, the regex engine would see «`\u00E0\d`». The regex engine doesn’t support the `\u` token. It will to match the literal text „`u00E0`” followed by a digit instead.

To avoid this confusion, just use Unicode raw strings like `ur"\u00E0\d"`. Then backslashes don’t need to be escaped. Python does interpret Unicode escapes in raw strings.

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 backreference 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)`.

20. How to Use Regular Expressions in REALbasic

REALbasicincludes a built-in RegEx class. Internally, this class is based on the open source PCRE library. What this means to you as a REALbasic developer is that the RegEx class provides you with a rich flavor of Perl-compatible regular expressions. The regular expression tutorial in this book does not explicitly mention REALbasic. Everything said in the tutorial about PCRE's regex flavor also applies to REALbasic. The only exception are the case insensitive and "multi-line" matching modes. In PCRE, they're off by default, while in REALbasic they're on by default.

REALbasic uses the UTF-8 version of PCRE. This means that if you want to process non-ASCII data that you've retrieved from a file or the network, you'll need to use REALbasic's TextConverter class to convert your strings into UTF-8 before passing them to the RegEx object. You'll also need to use the TextConverter to convert the strings returned by the RegEx class from UTF-8 back into the encoding your application is working with.

The RegEx Class

To use a regular expression, you need to create a new instance of the RegEx class. Assign your regular expression to the SearchPattern property. You can set various options in the Options property, which is an instance of the RegExOptions class.

To check if a regular expression matches a particular string, call the Search method of the RegEx object, and pass the subject string as a parameter. This method returns an instance of the RegExMatch class if a match is found, or Nil if no match is found. To find the second match in the same subject string, call the Search method again, without any parameters. Do not pass the subject string again, since doing so restarts the search from the beginning of the string. Keep calling Search without any parameters until it returns Nil to iterate over all regular expression matches in the string.

The RegExMatch Class

When the Regex.Search method finds a match, it stores the match's details in a RegExMatch object. This object has three properties. The SubExpressionCount property returns the number of capturing groups in the regular expression *plus one*. E.g. it returns 3 for the regex «(1)(2)». The SubExpressionString property returns the substring matched by the regular expression or a capturing group. SubExpressionString(0) returns the whole regex match, while SubExpressionString(1) through SubExpressionString(SubExpressionCount-1) return the matches of the capturing group. SubExpressionStartB returns the byte offset of the start of the match of the whole regex or one of the capturing groups depending on the numeric index you pass as a parameter to the property.

The RegExOptions Class

The RegExOptions class has nine properties to set various options for your regular expression.

- Set CaseSensitive (False by default) to True to treat uppercase and lowercase letters as different characters. This option is the *inverse* of "case insensitive mode" or `/i` in other programming languages.
- Set DotMatchAll (False by default) to True to make the dot match all characters, including line break characters. This option is the equivalent of "single line mode" or `/s` in other programming languages.
- Set Greedy (True by default) to False if you want quantifiers to be lazy, effectively making «`.*`» the same as «`.*?`». I strongly recommend against setting Greedy to False. Simply use the «`.*?`» syntax instead. This way, somebody reading your source code will clearly see when you're using greedy quantifiers and when you're using lazy quantifiers when they look only at the regular expression.
- The LineEndType option is the only one that takes an Integer instead of a Boolean. This option affect which character the caret and dollar treat as the "end of line" character. The default is 0, which accepts both `\r` and `\n` as end-of-line characters. Set it to 1 to use auto-detect the host platform, and use `\n` when your application runs on Windows and Linux, and `\r` when it runs on a Mac. Set it to 2 for Mac (`\r`), 3 for Windows (`\n`) and 4 for UNIX (`\n`). I recommend you leave this option as zero, which is most likely to give you the results you intended. This option is actually a modification to the PCRE library made in REALbasic. PCRE supports only option 4, which often confuses Windows developers since it causes «`test$`» to fail against "`test\r\n`" as Windows uses `\r\n` for line breaks.
- Set MatchEmpty (True by default) to False if you want to skip zero-length matches.
- Set ReplaceAllMatches (False by default) to True if you want the Regex.Replace method to search-and-replace all regex matches in the subject string rather than just the first one.
- Set StringBeginIsLineBegin (True by default) to False if you don't want the start of the string to be considered the start of the line. This can be useful if you're processing a large chunk of data as several separate strings, where only the first string should be considered as starting the (conceptual) overall string.
- Similarly, set StringEndIsLineEnd (True by default) to False if the string you're passing to the Search method isn't really the end of the whole chunk of data you're processing.
- Set TreatTargetAsOneLine (False by default) to make the caret and dollar match at the start and the end of the string only. By default, they will also match after and before embedded line breaks. This option is the *inverse* of the "multi-line mode" or `/m` in other programming languages.

REALbasic RegEx Source Code Example

```
'Prepare a regular expression object
Dim myRegEx As RegEx
Dim myMatch As RegExMatch
myRegEx = New RegEx
myRegEx.Options.TreatTargetAsOneLine = True
myRegEx.SearchPattern = "regex"
'Pop up all matches one by one
myMatch = myRegEx.Search(SubjectString)
While myMatch <> Nil
          MsgBox(myMatch.SubExpressionString(0))
          myMatch = myRegEx.Search()
Wend
```

Searching and Replacing

In addition to finding regex matches in a string, you can replace the matches with another string. To do so, set the ReplacementPattern property of your RegEx object, and then call the Replace method. Pass the source string as a parameter to the Replace method. The method will return a copy of the string with the replacement(s) applied. The RegEx.Options.ReplaceAllMatches property determines if only the first regex match or if all regex matches will be replaced.

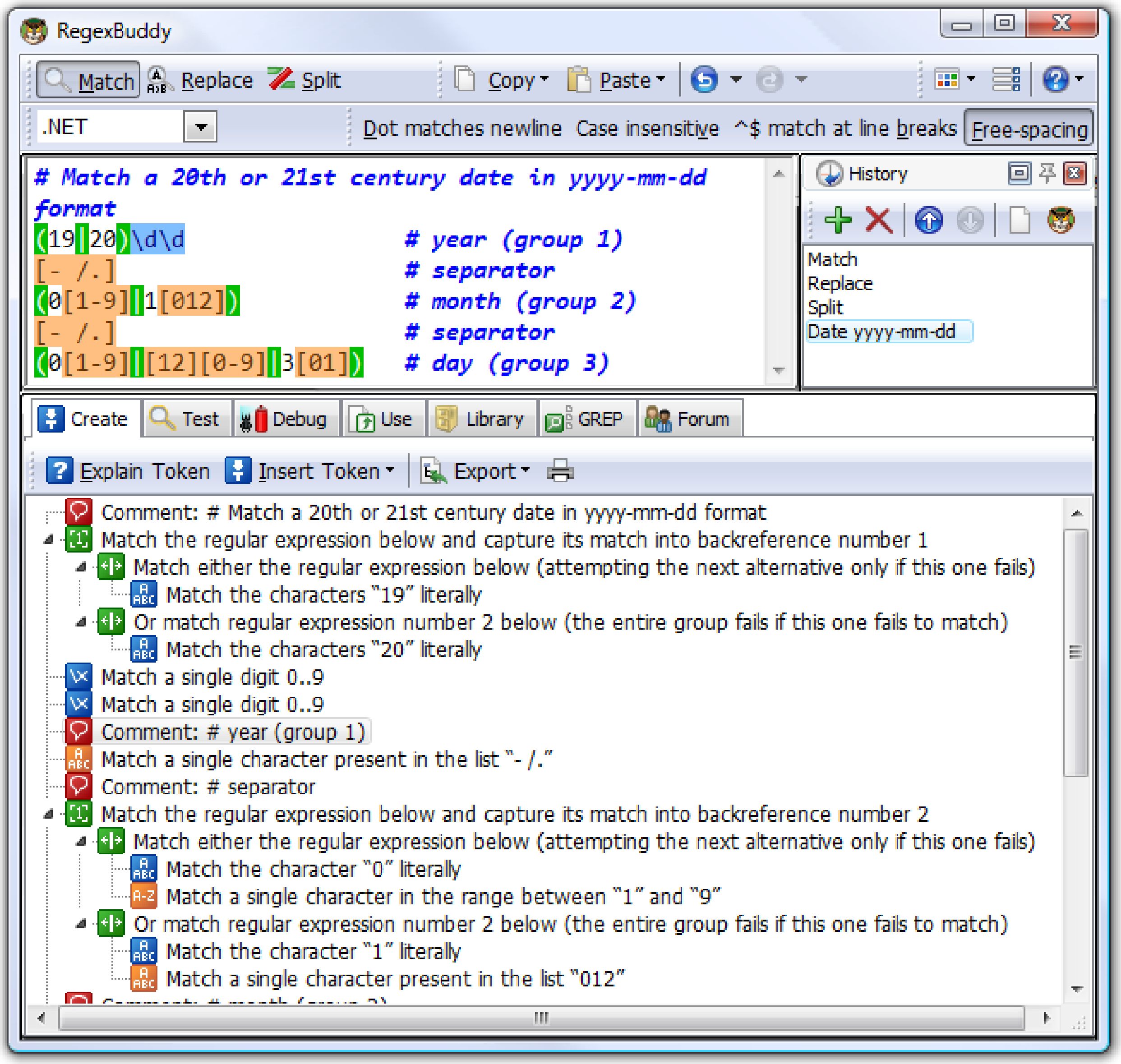
In the ReplacementPattern string, you can use `$&`, `$0` or `\0` to insert the whole regular expression match into the replacement. Use `$1` or `\1` for the match of the first capturing group, `$2` or `\2` for the second, etc.

If you want more control over how the replacements are made, you can iterate over the regex matches like in the code snippet above, and call the RegExMatch.Replace method for each match. This method is a bit of a misnomer, since it doesn't actually replace anything. Rather, it returns the RegEx.ReplacementPattern string with all references to the match and capturing groups substituted. You can use this results to make the replacements on your own. This method is also useful if you want to collect a combination of capturing groups for each regex match.

21. RegexBuddy: Your Perfect Companion for Working with Regular Expressions

Regular expressions remain a complex beast, even with a detailed regular expression tutorial at your disposal. RegexBuddy is a specialized tool that makes working with regular expressions much easier.

RegexBuddy lays out any regular expression in an easy-to-grasp tree of regex building blocks. RegexBuddy updates the tree as you edit the regular expression. Much easier is to work with the regex tree directly. Delete and move regex building blocks, and add new ones by selecting from clear descriptions. You can get a good overview of complex regular expressions by collapsing grouping and alternation blocks in the tree.



```
# Match a 20th or 21st century date in yyyy-mm-dd
format
(19|20)\d\d                    # year (group 1)
[- /.]                         # separator
(0[1-9]|1[012])                # month (group 2)
[- /.]                         # separator
(0[1-9]|[12][0-9]|3[01])       # day (group 3)
```

```
Comment: # Match a 20th or 21st century date in yyyy-mm-dd format
Match the regular expression below and capture its match into backreference number 1
   Match either the regular expression below (attempting the next alternative only if this one fails)
      Match the characters "19" literally
   Or match regular expression number 2 below (the entire group fails if this one fails to match)
      Match the characters "20" literally
Match a single digit 0..9
Match a single digit 0..9
Comment: # year (group 1)
Match a single character present in the list "- /."
Comment: # separator
Match the regular expression below and capture its match into backreference number 2
   Match either the regular expression below (attempting the next alternative only if this one fails)
      Match the character "0" literally
      Match a single character in the range between "1" and "9"
   Or match regular expression number 2 below (the entire group fails if this one fails to match)
      Match the character "1" literally
      Match a single character present in the list "012"
Comment: # month (group 2)
```

Interactive Regex Tester and Debugger

Even though RegexBuddy's regex tree makes it very clear how a regular expression works, the only way to be 100% sure whether a particular regex pattern does what you want is to test it. RegexBuddy provides a safe environment where you can interactively test and debug your regular expressions on sample text and files. RegexBuddy can highlight regex matches and capturing groups. The highlighting is automatically updated as you edit the regex, so you can instantly see the effects of your changes.

For detailed tests, RegexBuddy provides compete details about matches and capturing groups. You can easily test regex search-and-replace and split actions.

The key advantages of testing regular expressions with RegexBuddy are safety and speed. RegexBuddy cannot modify valuable files and actual data. You only see the effect would be. Opening a sample file or copying and pasting sample data to test a regular expression is much quicker than transferring the regex to the tool or source code you want to use it with, and creating your own test environment.

Quickly Develop Efficient Software

Many popular programming languages support regular expressions. If you are a programmer, using regular expressions enables you to do in a single or a handful lines of code what would otherwise require dozens or hundreds. When you use RegexBuddy, testing a single regular expression is far easier than debugging handwritten code that does the same. If others need to maintain your code later, they will benefit from RegexBuddy's regex analysis to quickly understand your code. You can insert RegexBuddy's regex tree as a comment in your source code.

RegexBuddy makes developing software with regexes even easier by providing you with auto-generated code snippets. Instead of remembering the correct classes and function calls, and how to represent a regex in source code, just tell RegexBuddy which language you are using and what you want to do. Copy and paste your custom-generated code snippet into your code editor, and run.

Using regular expressions not only saves you time. Unless you spend a lot of time hand-optimizing your own text searching and processing code, using regular expressions will speed up your software. This is certainly true if your language has a built-in regex engine that works at a lower level than your own code can.

Collect and Save Regular Expressions

Use RegexBuddy to collect your own library of handy regular expressions. You can save a regex with only one click. If you type in a brief description with each regex you store, RegexBuddy's regex lookup enables you to quickly find a previously saved regex that does what you want.

RegexBuddy also comes with a standard library of common regular expressions that you can use in a wide variety of situations.

Find out More and Get Your Own Copy of RegexBuddy

RegexBuddy works under Windows 98, ME, NT4, 2000, XP and Vista, as well as most versions Linux for Intel Pentium and AMD Athlon PCs. For more information on RegexBuddy, please visit www.regexbuddy.com. You will quickly earn the money you pay for RegexBuddy back many times over in the time and frustration you will save. RegexBuddy makes working with regular expressions much easier, quicker and efficient.

22. 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".
- `/x` tells Ruby to ignore whitespace between regex tokens.
- `/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.split(/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.

23. Tcl Has Three Regular Expression Flavors

Tcl 8.2 and later support three regular expression flavors. The Tcl man pages dub them Basic Regular Expressions (BRE), Extended Regular Expressions (ERE) and Advanced Regular Expressions (ARE). BRE and ERE are mainly for backward compatibility with previous versions of Tcl. These flavor implement the two flavors defined in the POSIX standard. AREs are new in Tcl 8.2. They're the default and recommended flavor. This flavor implements the POSIX ERE flavor, with a whole bunch of added features. Most of these features are inspired by similar features in Perl regular expressions.

Tcl's regular expression support is based on a library developed for Tcl by Henry Spencer. This library has since been used in a number of other programming languages and applications, such as the PostgreSQL database and the wxWidgets GUI library for C++. Everything said about Tcl in this regular expression tutorial applies to any tool that uses Henry Spencer's Advanced Regular Expressions.

There are a number of important differences between Tcl Advanced Regular Expressions and Perl-style regular expressions. Tcl uses «`\m`», «`\M`», «`\y`» and «`\Y`» for word boundaries. Perl and most other modern regex flavors use «`\b`» and «`\B`». In Tcl, these last two match a backspace and a backslash, respectively.

Tcl also takes a completely different approach to mode modifiers. The `(?letters)` syntax is the same, but the available mode letters and their meanings are quite different. Instead of adding mode modifiers to the regular expression, you can pass more descriptive switches like `-nocase` to the `regexp` and `regsub` commands for some of the modes. Mode modifier spans in the style of `(?modes:regex)` are not supported. Mode modifiers must appear at the start of the regex. They affect the whole regex. Mode modifiers in the regex override command switches. Tcl supports these modes:

- «`(?i)`» or `-nocase` makes the regex match case insensitive.
- «`(?c)`» makes the regex match case sensitive. This mode is the default.
- «`(?x)`» or `-expanded` activates the free-spacing regexp syntax.
- «`(?t)`» disables the free-spacing regexp syntax. This mode is the default. The "t" stands for "tight", the opposite of "expanded".
- «`(?b)`» tells Tcl to interpret the remainder of the regular expression as a Basic Regular Expression.
- «`(?e)`» tells Tcl to interpret the remainder of the regular expression as an Extended Regular Expression.
- «`(?q)`» tells Tcl to interpret the remainder of the regular expression as plain text. The "q" stands for "quoted".
- «`(?s)`» selects "non-newline-sensitive matching", which is the default. The "s" stands for "single line". In this mode, the dot and negated character classes will match all characters, including newlines. The caret and dollar will match only at the very start and end of the subject string.
- «`(?p)`» or `-linestop` enables "partial newline-sensitive matching". In this mode, the dot and negated character classes will not match newlines. The caret and dollar will match only at the very start and end of the subject string.
- «`(?w)`» or `-lineanchor` enables "inverse partial newline-sensitive matching". The "w" stands for "weird". (Don't look at me! I didn't come up with this.) In this mode, the dot and negated character classes will not match newlines. The caret and dollar will match after and before newlines.
- «`(?n)`» or `-line` enables what Tcl calls "newline-sensitive matching". The dot and negated character classes will not match newlines. The caret and dollar will match after and before newlines. Specifying «`(?n)`» or `-line` is the same as specifying «`(?pw)`» or `-linestop -lineanchor`.
- «`(?m)`» is a historical synonym for «`(?n)`». I recommend you never use it, to avoid confusion with Perl's «`(?m)`».

If you use regular expressions with Tcl and other programming languages, be careful when dealing with the newline-related matching modes. Tcl's designers found Perl's `/m` and `/s` modes confusing. They are confusing, but at least Perl has only two, and they both affect only one thing. In Perl, `/m` or «`(?m)`» enables "multi-line mode", which makes the caret and dollar match after and before newlines. By default, they match at the very start and end of the string only. In Perl, `/s` or «`(?s)`» enables "single line mode". This mode makes the dot match all characters, including line break. By default, it doesn't match line breaks. Perl does not have a mode modifier to exclude line breaks from negated character classes. In Perl, «`[^a]`» matches anything except "`a`", including newlines. The only way to exclude newlines is to write «`[^a\n]`». Perl's default matching mode is like Tcl's «`(?p)`», except for the difference in negated character classes.

Why compare Tcl with Perl? .NET, Java, PCRE and Python support the same «`(?m)`» and «`(?s)`» modifiers with the exact same defaults and effects as in Perl. JavaScript lacks `/s` and Ruby lacks `/m`, but at least they don't introduce completely different options. Negated character classes work the same in all these languages and libraries. It's unfortunate that Tcl didn't follow Perl's standard, since Tcl's four options are just as confusing as Perl's two options. Together they make a very nice alphabet soup.

If you ignore the fact that Tcl's options affect negated character classes, you can use the following table to translate between Tcl's newline modes and Perl-style newline modes. Note that the defaults are different. If you don't use any switches, «`(?s).`» and «`.`» are equivalent in Tcl, but not in Perl.

Tcl:	«`(?s)`» (default)
Perl:	«`(?s)`»
Dot:	Start and end of string only
Anchors:	Any character

Tcl:	«`(?p)`»
Perl:	(default)
Dot:	Start and end of string only
Anchors:	Any character except newlines

Tcl:	«`(?w)`»
Perl:	«`(?m)`»
Dot:	Start and end of string, and at newlines
Anchors:	Any character except newlines

Tcl:	«`(?n)`»
Perl:	«`(?sm)`»
Dot:	Start and end of string, and at newlines
Anchors:	Any character

Regular Expressions as Tcl Words

You can insert regular expressions in your Tcl source code either by enclosing them with double quotes (e.g. `"my regexp"`) or by enclosing them with curly braces (e.g. `{my regexp}`. Since the braces don't do any substitution like the quotes, they're by far the best choice for regular expressions.

The only thing you need to worry about is that unescaped braces in the regular expression must be balanced. Escaped braces don't need to be balanced, but the backslash used to escape the brace remains part of the regular expression. You can easily satisfy these requirements by escaping all braces in your regular expression, except those used as a quantifier. This way your regex will work as expected, and you don’t need to change it at all when pasting it into your Tcl source code, other than putting a pair of braces around it.

The regular expression «^\{\d{3}\\$» matches a string that consists entirely of an opening brace, three digits and one backslash. In Tcl, this becomes {^\{\d+{3}$\\}. There’s no doubling of backslashes or any sort of escaping needed, as long as you escape literal braces in the regular expression. «{» and «\{» are both valid regular expressions to match a single opening brace in a Tcl ARE (and any Perl-style regex flavor, for that matter). Only the latter will work correctly in a Tcl literal enclosed with braces.

Finding Regex Matches

It Tcl, you can use the regexp command to test if a regular expression matches (part of) a string, and to retrieve the matched part(s). The syntax of the command is:

regexp *?switches? regexp subject ?matchvar? ?group1var group2var ...?*

Immediately after the regexp command, you can place zero or more switches from the list above to indicate how Tcl should apply the regular expression. The only required parameters are the regular expression and the subject string. You can specify a literal regular expression using braces as I just explained. Or, you can reference any string variable holding a regular expression read from a file or user input.

If you pass the name of a variable as an additional argument, Tcl will store the part of the string matched by the regular expression into that variable. Tcl will *not* set the variable to an empty string if the match attempt fails. If the regular expressions has capturing groups, you can add additional variable names to capture the text matched by each group. If you specify fewer variables than the regex has capturing groups, the text matched by the additional groups is not stored. If you specify more variables than the regex has capturing groups, the additional variables will be set to an empty string if the overall regex match was successful.

The regexp command returns 1 if (part of) the string could be matched, and zero if there’s no match. The following script applies the regular expression «my regex» case insensitively to the string stored in the variable subjectstring and displays the result:

```
if [
  regexp -nocase {my regex} $subjectstring matchresult
] then {
  puts $matchresult
} else {
  puts “my regex could not match the subject string”
}
```

The regexp command supports three more switches that aren’t regex mode modifiers. The -all switch causes the command to return a number indicating how many times the regex could be matched. The variables storing the regex and group matches will store the last match in the string only.

The -inline switch tells the regexp command to return an array with the substring matched by the regular expression and all substrings matched by all capturing groups. If you also specify the -all switch, the array will contain the first regex match, all the group matches of the first match, then the second regex match, the group matches of the first match, etc.

The -start switch must be followed by a number (as a separate Tcl word) that indicates the character offset in the subject string at which Tcl should attempt the match. Everything before the starting position will be invisible to the regex engine. This means that «`\A`» will match at the character offset you specify with `-start`, even if that position is not at the start of the string.

Replacing Regex Matches

With the `regsub` command, you can replace regular expression matches in a string.

`regsub` *?switches? regexp replacement subject ?resultvar?*

Just like the `regexp` command, `regsub` takes zero or more switches followed by a regular expression. It supports the same switches, except for `-inline`. Remember to specify `-all` if you want to replace all matches in the string.

The argument after the regexp should be the replacement text. You can specify a literal replacement using the brace syntax, or reference a string variable. The `regsub` command recognizes a few metacharacters in the replacement text. You can use `\0` as a placeholder for the whole regex match, and `\1` through `\9` for the text matched by one of the first nine capturing groups. You can also use `&` as a synonym of `\0`. Note that there's no backslash in front of the ampersand. `&` is substituted with the whole regex match, while `\&` is substituted with a literal ampersand. Use `\\` to insert a literal backslash. You only need to escape backslashes if they're followed by a digit, to prevent the combination from being seen as a backreference. Again, to prevent unnecessary duplication of backslashes, you should enclose the replacement text with braces instead of double quotes. The replacement text `\1` becomes `{\1}` when using braces, and `"\\1"` when using quotes.

The final argument is optional. If you pass a variable reference as the final argument, that variable will receive the string with the replacements applied, and `regsub` will return an integer indicating the number of replacements made. If you omit the final argument, `regsub` will return the string with the replacements applied.

24. 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.

In fact, the regular expression flavor used in the version 5.5 VBScript object is the same one used by JavaScript and JScript. The regex flavor is part of the ECMA-262 standard for JavaScript. Therefore, everything said about JavaScript’s regular expression flavor in this book also applies to VBScript.

JavaScript and VBScript implement Perl-style regular expressions. However, they lack quite a number of advanced features available in Perl and other modern regular expression flavors:

- 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, except for matching single characters with `\uFFFF`
- No named capturing groups. Use numbered capturing groups instead.
- No mode modifiers to set matching options within the regular expression.
- No conditionals.
- 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

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.

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.

```
'Prepare a regular expression object
Set myRegExp = New RegExp
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.

25. 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>
```

26. 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 has decent regular expression capabilities starting with version 5.5. It implements the same regular expression flavor used in JavaScript, as standardized in the ECMA-262 standard for JavaScript. This library is part of Internet Explorer 5.5 and later. It is available on all computers running Windows XP or Vista, and previous versions of Windows if the user upgraded to IE 5.5 or later. That includes almost every Windows 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. Its 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. Anything said about JavaScript's flavor of regular expressions in the tutorial also applies to VBScript's flavor.

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
```

27. XML Schema Regular Expressions

The W3C XML Schema standard defines its own regular expression flavor. You can use it in the `pattern` facet of simple type definitions in your XML schemas. E.g. the following defines the simple type "SSN" using a regular expression to require the element to contain a valid US social security number.

```
<xsd:simpleType name="SSN">
    <xsd:restriction base="xsd:token">
        <xsd:pattern value="[0-9]{3}-[0-9]{2}-[0-9]{4}"/>
    </xsd:restriction>
</xsd:simpleType>
```

Compared with other regular expression flavors, the XML schema flavor is quite limited in features. Since it's only used to validate whether an entire element matches a pattern or not, rather than for extracting matches from large blocks of data, you won't really miss the features often found in other flavors. The limitations allow schema validators to be implemented with efficient text-directed engines.

Particularly noteworthy is the complete absence of anchors like the caret and dollar, word boundaries and lookaround. XML schema always implicitly anchors the entire regular expression. The regex must match the whole element for the element to be considered valid. If you have the pattern «`regexp`», the XML schema validator will apply it in the same way as say Perl, Java or .NET would do with the pattern «`^regexp$`». If you want to accept all elements with „`regex`" somewhere in the middle of their contents, you'll need to use the regular expression «`.*regex.*`». The to «`.*`» expand the match to cover the whole element, assuming it doesn't contain line breaks. If you want to allow line breaks, you can use something like «`[\s\S]*regex[\s\S]*`». Combining a shorthand character class with its negated version results in a character class that matches anything.

XML schemas do not provide a way to specify matching modes. The dot never matches line breaks, and patterns are always applied case sensitively. If you want to apply «`literal`» case insensitively, you'll need to rewrite it as «`[lL][iI][tT][eE][rR][aA][lL]`».

XML regular expressions don't have any tokens like `\xFF` or `\uFFFF` to match particular (non-printable) [illegible]

Lazy quantifiers are not available. Since the pattern is anchored at the start and the end of the subject string anyway, and only a success/failure result is returned, the only potential difference between a greedy and lazy quantifier would be performance. You can never make a fully anchored pattern match or fail by changing a greedy quantifier into a lazy one or vice versa.

XML regular expressions support the following:

- Character classes, including shorthands, ranges and negated classes.
- Character class subtraction.
- The dot, which matches any character except line breaks.
- Alternation and groups.
- Greedy quantifiers `?`, `*`, `+` and `{n,m}`
- Unicode properties and blocks

XML Character Classes

Despite its limitations, XML schema regular expressions introduce two handy features. The special shorthand character classes «\i» and «\c» make it easy to match XML names. No other regex flavor supports these.

Character class subtraction makes it easy to match a character that is in a certain list, but not in another list. E.g. «[a-z-[aeiou]]» matches an English consonant. This feature is now also available in the JGsoft and .NET regex engines. It is particularly handy when working with Unicode properties. E.g. «[\p{L}-[\p{IsBasicLatin}]]» matches any letter that is not an English letter.

Part 4

Reference

1. Basic Syntax Reference

Characters

Character: Any character except `[\^$.|?*+()`
Description: All characters except the listed special characters match a single instance of themselves. `{` and `}` are literal characters, unless they're part of a valid regular expression token (e.g. the `{n}` quantifier).
Example: «`a`» matches „`a`"

Character: `\` (backslash) followed by any of `[\^$.|?*+(){}`
Description: A backslash escapes special characters to suppress their special meaning.
Example: «`\+`» matches „`+`"

Character: `\Q...\E`
Description: Matches the characters between `\Q` and `\E` literally, suppressing the meaning of special characters.
Example: «`\Q+-*/\E`» 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: `\a`, `\e`, `\f` and `\v`
Description: Match a bell character (`\x07`), escape character (`\x1B`), form feed (`\x0C`) and vertical tab (`\x0B`) respectively. Can be used in character classes.
Character: `\cA` through `\cZ`
Description: Match an ASCII character Control+A through Control+Z, equivalent to «`\x01`» through «`\x1A`». Can be used in character classes.
Example: «`\cM\cJ`» 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

Character: `[\b]`
Description: Inside a character class, `\b` is a backspace character.
Example: «`[\b\t]`» matches a backspace or tab character

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: `(?x)`
Description: Turn on free-spacing mode to ignore whitespace between regex tokens, and allow # comments.
Character: `(?-x)`
Description: Turn off free-spacing mode.
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”

Character: `(?(1)then|else)`
Description: If the first capturing group took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the first capturing group did not take part in the match, the “else” part must match for the overall regex to match.
Example: «`(a)?(?(1)b|c)`» matches „`ab`”, the first „`c`” and the second „`c`” in “`babxcac`”

Comments

Character: `(?#comment)`
Description: Everything between `(?#` and `)` is ignored by the regex engine.
Example: «`a(?#foobar)b`» matches „`ab`”

3. 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 „©”

Character: `\x{FFFF}` where FFFF are 1 to 4 hexadecimal digits
Description: Perl syntax to match a specific Unicode code point. Can be used inside character classes.
Example: «`\x{E0}`» matches „à” encoded as U+00E0 only. «`\x{A9}`» matches „©”

Unicode Properties, Scripts and Blocks

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{Arabic}`
Description: Matches a single Unicode code point that is part of the Unicode script “Arabic”. See Unicode Scripts in the tutorial for a complete list of scripts. Each Unicode code point is part of exactly one script. Can be used inside character classes.
Example: «`\p{Thai}`» matches one of 83 code points in Thai script, from „ก” until „๙”

Character: `\p{InBasicLatin}`
Description: Matches a single Unicode code point that is part of the Unicode block “BasicLatin”. See Unicode Blocks in the tutorial for a complete list of blocks. Each Unicode code point is part of exactly one block. Blocks may contain unassigned code points. Can be used inside character classes.
Example: «`\p{InLatinExtended-A}`» any of the code points in the block U+100 until U+17F („Ā” until „ſ”)

Character: `\P{L}` or `\P{Letter}`
Description: Matches a single Unicode code point that does *not* have the property “letter”. You can also use \P to match a code point that is not part of a particular Unicode block or script. Can be used inside character classes.
Example: «`\P{L}`» matches „©”

4. 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”.

Character: (?(name)then|else)

Description: If the capturing group “name” took part in the match attempt thus far, the “then” part must match for the overall regex to match. If the capturing group “name” did not take part in the match, the “else” part must match for the overall regex to match.

Example: «(?<group>a)?(?(group)b|c)» matches „ab”, the first „c” and the second „c” in “babxcac”

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”.

XML Character Classes

Character: `\i`
Description: Matches any character that may be the first character of an XML name, i.e. «`[_:A-Za-z]`».
Character: `\c`
Description: «`\c`» matches any character that may occur after the first character in an XML name, i.e. «`[-._:A-Za-z0-9]`»
Example: «`\i\c*`» matches an XML name like „`xml:schema`”

Character: `\I`
Description: Matches any character that cannot be the first character of an XML name, i.e. «`[^_:A-Za-z]`».
Character: `\C`
Description: Matches any character that cannot occur in an XML name, i.e. «`[^-._:A-Za-z0-9]`».
Character: `[abc-[xyz]]`
Description: Subtracts character class “xyz” from character class “abc”. The result matches any single character that occurs in the character class “abc” but not in the character class “xyz”.
Example: «`[a-z-[aeiou]]`» matches any letter that is not a vowel (i.e. a consonant).

POSIX Bracket Expressions

Character: `[:alpha:]`
Description: Matches one character from a POSIX character class. Can only be used in a bracket expression.
Example: «`[[:digit:][:lower:]]`» matches one of „`0`” through „`9`” or „`a`” through „`z`”

Character: `[.span-ll.]`
Description: Matches a POSIX collation sequence. Can only be used in a bracket expression.
Example: «`[[.span-ll.]]`» matches „`ll`” in the Spanish locale

Character: `[=x=]`
Description: Matches a POSIX character equivalence. Can only be used in a bracket expression.
Example: «`[[=e=]]`» matches „`e`”, „`é`”, „`è`” and „`ê`” in the French locale

5. Regular Expression Flavor Comparison

The table below compares which regular expression flavors support which regex features and syntax. The features are listed in the same order as in the regular expression reference.

The comparison shows regular expression flavors rather than particular applications or programming languages implementing one of those regular expression flavors.

- JGsoft: This flavor is used by the JGsoft products, including PowerGREP and EditPad Pro.
- .NET: This flavor is used by programming languages based on the Microsoft .NET framework versions 1.x, 2.0 or 3.0. It is generally also the regex flavor used by applications developed in these programming languages.
- Java: The regex flavor of the java.util.regex package, available in the Java 4 (JDK 1.4.x) and later. A few features were added in Java 5 (JDK 1.5.x) and Java 6 (JDK 1.6.x). It is generally also the regex flavor used by applications developed in Java.
- Perl: The regex flavor used in the Perl programming language, as of version 5.8. Versions prior to 5.6 do not support Unicode.
- PCRE: The open source PCRE library. The feature set described here is available in PCRE 5.x and 6.x.
- ECMA (JavaScript): The regular expression syntax defined in the 3rd edition of the ECMA-262 standard, which defines the scripting language commonly known as JavaScript.
- Python: The regex flavor supported by Python's built-in `re` module.
- Ruby: The regex flavor built into the Ruby programming language.
- Tcl ARE: The regex flavor developed by Henry Spencer for the `regexp` command in Tcl 8.2 and 8.4, dubbed Advanced Regular Expressions.
- POSIX BRE: Basic Regular Expressions as defined in the IEEE POSIX standard 1003.2.
- POSIX ERE: Extended Regular Expressions as defined in the IEEE POSIX standard 1003.2.
- XML: The regular expression flavor defined in the XML Schema standard.

Applications and languages implementing one of the above flavors are:

- AceText: Version 2 and later use the JGsoft engine. Version 1 did not support regular expressions at all.
- C#: As a .NET programming language, C# can use the System.Text.RegularExpressions classes, listed as ".NET" below.
- Delphi for .NET: As a .NET programming language, the .NET version of Delphi can use the System.Text.RegularExpressions classes, listed as ".NET" below.
- Delphi for Win32: Delphi for Win32 does not have built-in regular expression support. Many free PCRE wrappers are available.
- EditPad Pro: Version 6 and later use the JGsoft engine. Earlier versions used PCRE, without Unicode support.
- egrep: The traditional UNIX egrep command uses the "POSIX ERE" flavor, though not all implementations fully adhere to the standard.
- grep: The traditional UNIX grep command uses the "POSIX BRE" flavor, though not all implementations fully adhere to the standard.
- Java: The regex flavor of the java.util.regex package is listed as "Java" in the table below.
- JavaScript: JavaScript's regex flavor is listed as "ECMA" in the table below.
- MySQL: MySQL uses POSIX Extended Regular Expressions, listed as "POSIX ERE" in the table below.

- Oracle: Oracle Database 10g implements POSIX Extended Regular Expressions, listed as “POSIX ERE” in the table below. Oracle supports backreferences `\1` through `\9`, though these are not part of the POSIX ERE standard.
- Perl: Perl’s regex flavor is listed as “Perl” in the table below.
- PHP: PHP’s `ereg` functions implement the “POSIX ERE” flavor, while the `preg` functions implement the “PCRE” flavor.
- PostgreSQL: PostgreSQL 7.4 and later uses Henry Spencer’s “Advanced Regular Expressions” flavor, listed as “Tcl ARE” in the table below. Earlier versions used POSIX Extended Regular Expressions, listed as POSIX ERE.
- PowerGREP: Version 3 and later use the JGsoft engine. Earlier versions used PCRE, without Unicode support.
- Python: Python’s regex flavor is listed as “Python” in the table below.
- REALbasic: REALbasic’s RegEx class is a wrapper around PCRE.
- RegexBuddy: Version 3 and later use a special version of the JGsoft engine that emulates all the regular expression flavors in this comparison. Version 2 supported the JGsoft regex flavor only. Version 1 used PCRE, without Unicode support.
- Ruby: Ruby’s regex flavor is listed as “Ruby” in the table below.
- Tcl: Tcl’s Advanced Regular Expression flavor, the default flavor in Tcl 8.2 and later, is listed as “Tcl ARE” in the table below. Tcl’s Extended Regular Expression and Basic Regular Expression flavors are listed as “POSIX ERE” and “POSIX BRE” in the table below.
- VBScript: VBScript’s RegExp object uses the same regex flavor as JavaScript, which is listed as “ECMA” in the table below.
- Visual Basic 6: Visual Basic 6 does not have built-in support for regular expressions, but can easily use the "Microsoft VBScript Regular Expressions 5.5" COM object, which implements the “ECMA” flavor listed below.
- Visual Basic.NET: As a .NET programming language, VB.NET can use the System.Text.RegularExpressions classes, listed as ".NET" below.
- XML: The XML Schema regular expression flavor is listed as “XML” in the table below.

Characters

Feature: Backslash escapes one metacharacter
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, XML

Feature: `\Q`...`\E` escapes a string of metacharacters
Supported by: JGsoft, Java, Perl, PCRE

Feature: `\x00` through `\xFF` (ASCII character)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE

Feature: `\n` (LF), `\r` (CR) and `\t` (tab)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, XML

Feature: `\f` (form feed) and `\v` (vtab)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE

Feature: `\a` (bell) and `\e` (escape)
Supported by: JGsoft, .NET, Java, Perl, PCRE, Python, Ruby, Tcl ARE

Feature: `\b` (backspace) and `\B` (backslash)
Supported by: Tcl ARE

Feature: `\cA` through `\cZ` (control character)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Tcl ARE

Feature: `\ca` through `\cz` (control character)
Supported by: JGsoft, .NET, Perl, PCRE, JavaScript, Tcl ARE

Character Classes or Character Sets

Feature: `[abc]` character class
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, XML

Feature: `[a-z]` character class range
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, XML

Feature: `[^abc]` negated character class
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, XML

Feature: Backslash escapes one character class metacharacter
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, XML

Feature: \Q...\E escapes a string of character class metacharacters
Supported by: JGsoft, Java, Perl, PCRE

Feature: `\d`, `\w` and `\s` shorthand character classes
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, XML

Feature: `\D`, `\W` and `\S` shorthand negated character classes
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, XML

Feature: `[\b]` backspace
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE

Dot

Feature: `.` (dot; any character except line break)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, XML

Anchors

Feature: `^` (start of string/line)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE

Feature: `$` (end of string/line)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE

Feature: `\A` (start of string)
Supported by: JGsoft, .NET, Java, Perl, PCRE, Python, Ruby, Tcl ARE

Feature: `\Z` (end of string, before final line break)
Supported by: JGsoft, .NET, Java, Perl, PCRE, Python, Ruby, Tcl ARE

Feature: `\z` (end of string)
Supported by: JGsoft, .NET, Java, Perl, PCRE, Ruby

Word Boundaries

Feature: `\b` (at the beginning or end of a word)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby

Feature: `\B` (NOT at the beginning or end of a word)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby

Feature: `\y` (at the beginning or end of a word)
Supported by: JGsoft, Tcl ARE

Feature: `\Y` (NOT at the beginning or end of a word)
Supported by: JGsoft, Tcl ARE

Feature: `\m` (at the beginning of a word)
Supported by: JGsoft, Tcl ARE

Feature: `\M` (at the end of a word)
Supported by: JGsoft, Tcl ARE

Alternation

Feature: `|` (alternation)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX ERE, XML

Quantifiers

Feature: `?` (0 or 1)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX ERE, XML

Feature: `*` (0 or more)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, XML

Feature: `+` (1 or more)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX ERE, XML

Feature: `{n}` (exactly n)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, XML

Feature: `{n,m}` (between n and m)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, XML

Feature: `{n,}` (n or more)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, XML

Feature: `?` after any of the above quantifiers to make it “lazy”
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE

Grouping and Backreferences

Feature: `(regex)` (numbered capturing group)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, XML

Feature: `(?:regex)` (non-capturing group)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE

Feature: `\1` through `\9` (backreferences)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE

Feature: `\10` through `\99` (backreferences)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE

Modifiers

Feature: `(?i)` (case insensitive)
Supported by: JGsoft, .NET, Java, Perl, PCRE, Python, Ruby, Tcl ARE

Feature: `(?s)` (dot matches newlines)
Supported by: JGsoft, .NET, Java, Perl, PCRE, Python, Ruby

Feature: `(?m)` (`^` and `$` match at line breaks)
Supported by: JGsoft, .NET, Java, Perl, PCRE, Python

Feature: `(?x)` (free-spacing mode)
Supported by: JGsoft, .NET, Java, Perl, PCRE, Python, Ruby, Tcl ARE

Feature: `(?n)` (explicit capture)
Supported by: JGsoft, .NET

Feature: `(?-ismxn)` (turn off mode modifiers)
Supported by: JGsoft, .NET, Java, Perl, PCRE, Ruby

Feature: `(?ismxn:group)` (mode modifiers local to group)
Supported by: JGsoft, .NET, Java, Perl, PCRE, Ruby

Atomic Grouping and Possessive Quantifiers

Feature: `(?>regex)` (atomic group)
Supported by: JGsoft, .NET, Java, Perl, PCRE, Ruby

Feature: `?+`, `*+`, `++` and `{m,n}+` (possessive quantifiers)
Supported by: JGsoft, Java, PCRE

Lookaround

Feature: `(?=regex)` (positive lookahead)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE

Feature: `(?!regex)` (negative lookahead)
Supported by: JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE

Feature: `(?<=text)` (positive lookbehind)
Supported by: JGsoft, .NET, Java, Perl, PCRE, Python

Feature: `(?<!text)` (negative lookbehind)
Supported by: JGsoft, .NET, Java, Perl, PCRE, Python

Continuing from The Previous Match

Feature: `\G` (start of match attempt)
Supported by: JGsoft, .NET, Java, Perl, PCRE, Ruby

Conditionals

Feature: `(?(?=regex)then|else)`
Supported by: JGsoft, .NET, Perl, PCRE

Feature: `(?(1)then|else)`
Supported by: JGsoft, .NET, Perl, PCRE, Python

Feature: `(?(group)then|else)`
Supported by: JGsoft, .NET, PCRE, Python

Comments

Feature: `(?#comment)`
Supported by: JGsoft, .NET, Perl, PCRE, Python, Ruby, Tcl ARE

Unicode Characters

Feature: `\X` (Unicode grapheme)
Supported by: JGsoft, Perl, PCRE

Feature: `\u0000` through `\uFFFF` (Unicode character)
Supported by: JGsoft, .NET, Java, JavaScript, Python, Tcl ARE

Feature: `\x{0}` through `\x{FFFF}` (Unicode character)
Supported by: JGsoft, Perl, PCRE

Unicode Properties, Scripts and Blocks

Feature: `\pL` through `\pC` (Unicode properties)
Supported by: JGsoft, Java, Perl, PCRE

Feature: `\p{L}` through `\p{C}` (Unicode properties)
Supported by: JGsoft, .NET, Java, Perl, PCRE, XML

Feature: `\p{Lu}` through `\p{Cn}` (Unicode property)
Supported by: JGsoft, .NET, Java, Perl, PCRE, XML

Feature: `\p{L&}` and `\p{Letter&}` (equivalent of `[\p{Lu}\p{Ll}\p{Lt}]` Unicode properties)
Supported by: JGsoft, Perl, PCRE

Feature: `\p{IsL}` through `\p{IsC}` (Unicode properties)
Supported by: JGsoft, Java, Perl

Feature: `\p{IsLu}` through `\p{IsCn}` (Unicode property)
Supported by: JGsoft, Java, Perl

Feature: `\p{Letter}` through `\p{Other}` (Unicode properties)
Supported by: JGsoft, Perl

Feature: `\p{Lowercase_Letter}` through `\p{Not_Assigned}` (Unicode property)
Supported by: JGsoft, Perl

Feature: `\p{IsLetter}` through `\p{IsOther}` (Unicode properties)
Supported by: JGsoft, Perl

Feature: `\p{IsLowercase_Letter}` through `\p{IsNot_Assigned}` (Unicode property)
Supported by: JGsoft, Perl

Feature: `\p{Arabic}` through `\p{Yi}` (Unicode script)
Supported by: JGsoft, Perl, PCRE

Feature: `\p{IsArabic}` through `\p{IsYi}` (Unicode script)
Supported by: JGsoft, Perl

Feature: `\p{BasicLatin}` through `\p{Specials}` (Unicode block)
Supported by: JGsoft, Perl

Feature: `\p{InBasicLatin}` through `\p{InSpecials}` (Unicode block)
Supported by: JGsoft, Java, Perl

Feature: `\p{IsBasicLatin}` through `\p{IsSpecials}` (Unicode block)
Supported by: JGsoft, .NET, Perl, XML

Feature: Part between `{}` in all of the above is case insensitive
Supported by: JGsoft, Perl

Feature: Spaces, hyphens and underscores allowed in all long names listed above (e.g. BasicLatin can be written as Basic-Latin or Basic_Latin or Basic Latin)
Supported by: JGsoft, Java, Perl

Feature: `\P` (negated variants of all `\p` as listed above)
Supported by: JGsoft, .NET, Java, Perl, PCRE, XML

Feature: `\p{^...}` (negated variants of all `\p{...}` as listed above)
Supported by: JGsoft, Perl, PCRE

.NET Syntax for Named Capture and Backreferences

Feature: `(?<name>regex)` (named capturing group)
Supported by: JGsoft, .NET

Feature: `(?'name'regex)` (named capturing group)
Supported by: JGsoft, .NET

Feature: `\k<name>` (named backreference)
Supported by: JGsoft, .NET

Feature: `\k'name'` (named backreference)
Supported by: JGsoft, .NET

Python Syntax for Named Capture and Backreferences

Feature: `(?P<name>regex)` (named capturing group
Supported by: JGsoft, PCRE, Python

Feature: `(?P=name)` (named backreference)
Supported by: JGsoft, PCRE, Python

XML Character Classes

Feature: `\i`, `\I`, `\c` and `\C` shorthand XML name character classes
Supported by: XML

Feature: `[abc-[abc]]` character class subtraction
Supported by: JGsoft, .NET, XML

POSIX Bracket Expressions

Feature: `[:alpha:]` POSIX character class
Supported by: JGsoft, Perl, PCRE, Ruby, Tcl ARE, POSIX BRE, POSIX ERE

Feature: `\p{Alpha}` POSIX character class
Supported by: Java

Feature: `\P{Alpha}` negated POSIX character class
Supported by: Java

Feature: `[.span-ll.]` POSIX collation sequence
Supported by: Tcl ARE, POSIX BRE, POSIX ERE

Feature: `[=x=]` POSIX character equivalence
Supported by: Tcl ARE, POSIX BRE, POSIX ERE

6. Replacement Text Reference

The table below compares the various tokens that the various tools and languages discussed in this book recognize in the replacement text during search-and-replace operations.

The list of replacement text flavors is not the same as the list of regular expression flavors in the regex features comparison. The reason is that the replacements are not made by the regular expression engine, but by the tool or programming library providing the search-and-replace capability. The result is that tools or languages using the same regex engine may behave differently when it comes to making replacements. E.g. The PCRE library does not provide a search-and-replace function. All tools and languages implementing PCRE use their own search-and-replace feature, which may result in differences in the replacement text syntax. So these are listed separately.

To make the table easier to read, I did group tools and languages that use the exact same replacement text syntax. The labels for the replacement text flavors are only relevant in the table below. E.g. the .NET framework does have built-in search-and-replace function in its Regex class, which is used by all tools and languages based on the .NET framework. So these are listed together under ".NET".

Note that the escape rules below only refer to the replacement text syntax. If you type the replacement text in an input box in the application you're using, or if you retrieve the replacement text from user input in the software you're developing, these are the only escape rules that apply. If you pass the replacement text as a literal string in programming language source code, you'll need to apply the language's string escape rules on top of the replacement text escape rules.

A flavor can have four levels of support (or non-support) for a particular token. A "YES" in the table below indicates the token will be substituted. A "no" indicates the token will remain in the replacement as literal text. Note that languages that use variable interpolation in strings may still replace tokens indicated as unsupported below, if the syntax of the token corresponds with the variable interpolation syntax. E.g. in Perl, `$0` is replaced with the name of the script. Finally, "error" indicates the token will result in an error condition or exception, preventing any replacements being made at all.

- JGsoft: This flavor is used by the JGsoft products, including PowerGREP, EditPad Pro and AceText.
- .NET: This flavor is used by programming languages based on the Microsoft .NET framework versions 1.x, 2.0 or 3.0. It is generally also the regex flavor used by applications developed in these programming languages.
- Java: The regex flavor of the java.util.regex package, available in the Java 4 (JDK 1.4.x) and later. A few features were added in Java 5 (JDK 1.5.x) and Java 6 (JDK 1.6.x). It is generally also the regex flavor used by applications developed in Java.
- Perl: The regex flavor used in the Perl programming language, as of version 5.8.
- ECMA (JavaScript): The regular expression syntax defined in the 3rd edition of the ECMA-262 standard, which defines the scripting language commonly known as JavaScript. The VBscript RegExp object, which is also commonly used in VB 6 applications uses the same implementation with the same search-and-replace features.
- Python: The regex flavor supported by Python's built-in `re` module.
- Ruby: The regex flavor built into the Ruby programming language.
- Tcl: The regex flavor used by the `regsub` command in Tcl 8.2 and 8.4, dubbed Advanced Regular Expressions in the Tcl man pages.
- PHP ereg: The replacement text syntax used by the ereg_replace and eregi_replace functions in PHP.
- PHP preg: The replacement text syntax used by the preg_replace function in PHP.

- REALbasic: The replacement text syntax used by the ReplaceText property of the RegEx class in REALbasic.
- Oracle: The replacement text syntax used by the REGEXP_REPLACE function in Oracle Database 10g.
- Postgres: The replacement text syntax used by the regexp_replace function in PostgreSQL.

Syntax Using Backslashes

Feature: `\&` (whole regex match)
Supported by: JGsoft, Ruby, Postgres

Feature: `\0` (whole regex match)
Supported by: JGsoft, Python, Ruby, Tcl, PHP ereg, PHP preg, REALbasic

Feature: `\1` through `\9` (backreference)
Supported by: JGsoft, Perl, Python, Ruby, Tcl, PHP ereg, PHP preg, REALbasic, Oracle, Postgres

Feature: `\10` through `\99` (backreference)
Supported by: JGsoft, Python, PHP preg, REALbasic

Feature: `\10` through `\99` treated as `\1` through `\9` (and a literal digit) if fewer than 10 groups
Supported by: JGsoft

Feature: `\g<group>` (named backreference)
Supported by: JGsoft, Python

Feature: ``\` `` (backtick; subject text to the left of the match)
Supported by: JGsoft, Ruby

Feature: `\'` (straight quote; subject text to the right of the match)
Supported by: JGsoft, Ruby

Feature: `\+` (highest-numbered participating group)
Supported by: JGsoft, Ruby

Feature: Backslash escapes one backslash and/or dollar
Supported by: JGsoft, Java, Perl, Python, Ruby, Tcl, PHP ereg, PHP preg, REALbasic, Oracle, Postgres

Feature: Unescaped backslash as literal text
Supported by: JGsoft, .NET, Perl, JavaScript, Python, Ruby, Tcl, PHP ereg, PHP preg, REALbasic, Oracle, Postgres

Syntax Using Dollar Signs

Feature: `$&` (whole regex match)
Supported by: JGsoft, .NET, Perl, JavaScript, REALbasic

Feature: `$0` (whole regex match)
Supported by: JGsoft, .NET, Java, PHP preg, REALbasic

Feature: `$1` through `$9` (backreference)
Supported by: JGsoft, .NET, Java, Perl, JavaScript, PHP preg, REALbasic

Feature: `$10` through `$99` (backreference)
Supported by: JGsoft, .NET, Java, Perl, JavaScript, PHP preg, REALbasic

Feature: `$10` through `$99` treated as `$1` through `$9` (and a literal digit) if fewer than 10 groups
Supported by: JGsoft, Java, JavaScript

Feature: `${1}` through `${99}` (backreference)
Supported by: JGsoft, .NET, Perl, PHP preg

Feature: `${group}` (named backreference)
Supported by: JGsoft, .NET

Feature: ``$` `` (backtick; subject text to the left of the match)
Supported by: JGsoft, .NET, Perl, JavaScript, REALbasic

Feature: `$'` (straight quote; subject text to the right of the match)
Supported by: JGsoft, .NET, Perl, JavaScript, REALbasic

Feature: `$_` (entire subject string)
Supported by: JGsoft, .NET, Perl, JavaScript

Feature: `$+` (highest-numbered participating group)
Supported by: JGsoft, Perl

Feature: `$+` (highest-numbered group in the regex)
Supported by: .NET, JavaScript

Feature: `$$` (escape dollar with another dollar)
Supported by: JGsoft, .NET, JavaScript

Feature: `$` (unescaped dollar as literal text)
Supported by: JGsoft, .NET, JavaScript, Python, Ruby, Tcl, PHP ereg, PHP preg, Oracle, Postgres

Tokens Without a Backslash or Dollar

Feature: `&` (whole regex match)
Supported by: Tcl

General Replacement Text Behavior

Feature:	Backreferences to non-existent groups are silently removed
Supported by:	JGsoft, Perl, Ruby, Tcl, PHP preg, REALbasic, Oracle, Postgres

Highest-Numbered Capturing Group

The `$+` token is listed twice, because it doesn't have the same meaning in the languages that support it. It was introduced in Perl, where the `$+` variable holds the text matched by the highest-numbered capturing group that actually participated in the match. In several languages and libraries that intended to copy this feature, such as .NET and JavaScript, `$+` is replaced with the highest-numbered capturing group, whether it participated in the match or not.

E.g. in the regex «`a(\d)|x(\w)`» the highest-numbered capturing group is the second one. When this regex matches „`a4`", the first capturing group matches „`4`", while the second group doesn't participate in the match attempt at all. In Perl, `$+` will hold the „`4`" matched by the first capturing group, which is the highest-numbered group that actually participated in the match. In .NET or JavaScript, `$+` will be substituted with nothing, since the highest-numbered group in the regex didn't capture anything. When the same regex matches „`xy`", Perl, .NET and JavaScript will all store „`y`" in `$+`.

Also note that .NET numbers named capturing groups after all non-named groups. This means that in .NET, `$+` will always be substituted with the text matched by the last named group in the regex, whether it is followed by non-named groups or not, and whether it actually participated in the match or not.

Index