Part 6

Regular Expression Reference

1. Basic Syntax Reference

Characters

Character:	Any character except [\^$.|?*+()
Description:	All characters except the listed special characters match a single instance of themselves. { 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 for a few character escapes that are indicated with "can be used inside character classes".
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, word characters (letters, digits, and underscores), and whitespace (spaces, tabs, and line breaks). 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 >= 0 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 „aaaa”, „aaa” or „aa”
Character:	{n,m}? where n >= 0 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 „aa”, „aaa” or „aaaa”
Character:	{n,} where n >= 0
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 >= 0
Description:	Repeats the previous item n or more 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:	(?<=regex)
Description:	Zero-width positive lookbehind. Matches at a position if the pattern inside the lookahead can be matched ending at that position (i.e. to the left of that position). Depending on the regex flavor you're using, you may not be able to use quantifiers and/or alternation inside lookbehind.
Example:	«(?<=s)t» matches the first „t" in „streets".

Character:	(?<!regex)
Description:	Zero-width negative lookbehind. Matches at a position if the pattern inside the lookahead cannot be matched ending at 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 „๛"
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 Just Great Software 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.x. 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, versions 5.6 and 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. PCRE is the regex engine used by the TPerlRegEx Delphi component and the RegularExrpessions and RegularExpressionsCore units in Delphi XE and C++Builder XE.
- 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.
- GNU BRE: GNU Basic Regular Expressions, which are POSIX BRE with GNU extensions, used in the GNU implementations of classic UNIX tools.
- GNU ERE: GNU Extended Regular Expressions, which are POSIX ERE with GNU extensions, used in the GNU implementations of classic UNIX tools.
- XML: The regular expression flavor defined in the XML Schema standard.
- XPath: The regular expression flavor defined in the XQuery 1.0 and XPath 2.0 Functions and Operators 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.
- ActionScript: ActionScript is the language for programming Adobe Flash (formerly Macromedia Flash). ActionScript 3.0 and later supports the regex flavor listed as "ECMA" in the table below.
- awk: The awk UNIX tool and programming language uses POSIX ERE.
- 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. Linux usually ships with the GNU implementation, which use "GNU ERE".
- grep: The traditional UNIX grep command uses the "POSIX BRE" flavor, though not all implementations fully adhere to the standard. Linux usually ships with the GNU implementation, which use "GNU BRE".
- Emacs: The GNU implementation of this classic UNIX text editor uses the "GNU ERE" flavor, except that POSIX classes, collations and equivalences are not supported.
- 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.
- PowerShell: PowerShell's built-in -match and -replace operators use the .NET regex flavor. PowerShell can also use the System.Text.RegularExpressions classes directly.
- Python: Python's regex flavor is listed as "Python" in the table below.
- R: The regular expression functions in the R language for statistical programming use either the POSIX ERE flavor (default), the PCRE flavor (perl = true) or the POSIX BRE flavor (perl = false, extended = false).
- 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.
- sed: The sed UNIX tool uses POSIX BRE. Linux usually ships with the GNU implementation, which use "GNU BRE".
- 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.

- wxWidgets: The wxRegEx class supports 3 flavors. wxRE_ADVANCED is the “Tcl ARE” flavor, wxRE_EXTENDED is “POSIX ERE” and wxRE_BASIC is “POSIX BRE”.
- XML Schema: The XML Schema regular expression flavor is listed as “XML” in the table below.
- XPath: The regex flavor used by XPath functions is listed as “XPath” in the table below.
- XQuery: The regex flavor used by XQuery functions is listed as “XPath” 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, GNU BRE, GNU ERE, XML, XPath
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, XPath
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 [abc]

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

Feature:	[a-z] character class range
Supported by:	JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath
Feature:	Hyphen in [\d-z] is a literal
Supported by:	JGsoft, .NET, Java, Perl, PCRE
Feature:	Hyphen in [a-\d] is a literal
Supported by:	JGsoft, PCRE
Feature:	Backslash escapes one character class metacharacter
Supported by:	JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, XML, XPath
Feature:	\Q...\E escapes a string of character class metacharacters
Supported by:	JGsoft, Java, Perl, PCRE
Feature:	\d shorthand for digits
Supported by:	JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, XML, XPath
Feature:	\w shorthand for word characters
Supported by:	JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, GNU BRE, GNU ERE, XML, XPath
Feature:	\s shorthand for whitespace
Supported by:	JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, GNU BRE, GNU ERE, XML, XPath
Feature:	\D, \W and \S shorthand negated character classes
Supported by:	JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, GNU BRE, GNU ERE, XML, XPath
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, GNU BRE, GNU ERE, XML, XPath

Anchors

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

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

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

Feature: \` (start of string)
Supported by: GNU BRE, GNU ERE

Feature: \' (end of string)
Supported by: GNU BRE, GNU ERE

Word Boundaries

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

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

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

Feature: \< (at the beginning of a word)
Supported by: GNU BRE, GNU ERE

Feature: \> (at the end of a word)
Supported by: GNU BRE, GNU ERE

Alternation

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

Quantifiers

Feature:	? (0 or 1)
Supported by:	JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath
Feature:	* (0 or more)
Supported by:	JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath
Feature:	+ (1 or more)
Supported by:	JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath
Feature:	{n} (exactly n)
Supported by:	JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath
Feature:	{n,m} (between n and m)
Supported by:	JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath
Feature:	{n,} (n or more)
Supported by:	JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU ERE, XML, XPath
Feature:	? after any of the above quantifiers to make it "lazy"
Supported by:	JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, XPath

Grouping and Backreferences

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

Feature:	\10 through \99 (backreferences)
Supported by:	JGsoft, .NET, Java, Perl, PCRE, JavaScript, Python, Ruby, Tcl ARE, XPath
Feature:	Forward references \1 through \9
Supported by:	JGsoft, .NET, Java, Perl, PCRE, Ruby
Feature:	Nested references \1 through \9
Supported by:	JGsoft, .NET, Java, Perl, PCRE, JavaScript, Ruby
Feature:	Backreferences non-existent groups are an error
Supported by:	JGsoft, .NET, Java, Perl, PCRE, Python, Tcl ARE, POSIX BRE, GNU BRE, GNU ERE, XPath
Feature:	Backreferences to failed groups also fail
Supported by:	JGsoft, .NET, Java, Perl, PCRE, Python, Ruby, Tcl ARE, POSIX BRE, GNU BRE, GNU ERE, XPath

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) (using any lookaround)
Supported by:	JGsoft, .NET, Perl, PCRE
Feature:	(?(regex)then|else)
Supported by:	.NET
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

Free-Spacing Syntax

Feature:	Free-spacing syntax supported
Supported by:	JGsoft, .NET, Java, Perl, PCRE, Python, Ruby, Tcl ARE, XPath

Feature:	Character class is a single token
Supported by:	JGsoft, .NET, Perl, PCRE, Python, Ruby, Tcl ARE, XPath

Feature:	# starts a comment
Supported by:	JGsoft, .NET, Java, 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, XPath

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

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, XPath

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, XPath

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

Named Capture and Backreferences

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

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

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

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

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

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

Feature:	multiple capturing groups can have the same name
Supported by:	JGsoft, .NET

XML Character Classes

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

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

POSIX Bracket Expressions

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

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

Feature:	\p{IsAlpha} POSIX character class
Supported by:	JGsoft, Perl

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

Feature:	[=x=] POSIX character equivalence
Supported by:	Tcl ARE, POSIX BRE, POSIX ERE, GNU BRE, GNU 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. E.g. for languages that require backslashes in string literals to be escaped, you'll need to use "\\1“ instead of ”\1" to get the first backreference.

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.
- The "string" label indicates that the syntax is supported by string literals in the language's source code. For languages like PHP that have interpolated (double quotes) and non-interpolated (single quotes) variants, you'll need to use the interpolated string style. String-level support also means that the character escape won't be interpreted for replacement text typed in by the user or read from a file.
- 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 Just Great Software products, including PowerGREP, EditPad Pro and AceText. It is also used by the TPerlRegEx Delphi component and the RegularExrpessions and RegularExpressionsCore units in Delphi XE and C++Builder XE.
- .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. However, VBScript and VB strings don't support \xFF and \uFFFF escapes.
- 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. wxWidgets uses the same flavor.
- 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.
- XPath: The replacement text syntax used by the fn:replace function in XQuery and XPath.
- R: The replacement text syntax used by the sub and gsub functions in the R language. Though R supports three regex flavors, it has only one replacement syntax for all three.

Syntax Using Backslashes

Feature:	\& (whole regex match)
Supported by:	JGsoft, Ruby, Postgres
Feature:	\0 (whole regex match)
Supported by:	JGsoft, 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, R
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, XPath, R
Feature:	Unescaped backslash as literal text
Supported by:	JGsoft, .NET, Perl, JavaScript, Python, Ruby, Tcl, PHP ereg, PHP preg, Oracle, Postgres

Character Escapes

Feature:	\u0000 through \uFFFF (Unicode character)
Supported by:	JGsoft, Java, JavaScript, Python, Tcl, R
Feature:	\x{0} through \x{FFFF} (Unicode character)
Supported by:	JGsoft, Perl
Feature:	\x00 through \xFF (ASCII character)
Supported by:	JGsoft, Perl, JavaScript, Python, Tcl, PHP ereg, PHP preg, REALbasic, R

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, XPath
Feature:	$1 through $9 (backreference)
Supported by:	JGsoft, .NET, Java, Perl, JavaScript, PHP preg, REALbasic, XPath
Feature:	$10 through $99 (backreference)
Supported by:	JGsoft, .NET, Java, Perl, JavaScript, PHP preg, REALbasic, XPath
Feature:	$10 through $99 treated as $1 through $9 (and a literal digit) if fewer than 10 groups
Supported by:	JGsoft, Java, JavaScript, XPath
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
Feature:	$+ (highest-numbered participating group)
Supported by:	JGsoft, Perl
Feature:	$+ (highest-numbered group in the regex)
Supported by:	.NET
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, R

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, XPath

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