APPENDIX A

Regular Expression Reference

THIS APPENDIX PROVIDES a comprehensive quick reference for your day-to-day Java regular expression needs. The material in this appendix is presented as pure regex patterns, not the Java `String`-delimited counterparts. For example, when referring to a digit, ***\d***, not ***\\d***, is used here.

Table A-1. Common Characters

Regex	Description	Notes
q	The character *q*	Could be any character, not including special regex characters or punctuation.
\\	The backslash (**) character	\ delimits special regex characters, of which the backslash is a member.
\t	The tab character	
\n	The newline or linefeed character	
\r	The carriage-return character	
\f	The form-feed character	

Table A-2. Predefined Character Classes

Regex	Description	Notes
.	Any single character	Matches any single character, including spaces, punctuation, letters, or numbers. It may or may not match line-termination characters, depending on the operating system involved, or if the DOTALL flag is active in the regex pattern. Thus, it's probably a good idea to explicitly set, or turn off, DOTALL support in your patterns, in case you need to port your code.
\d	Any single digit from 0 to 9	
\D	Will match any character except a single digit	By default, this won't match line terminators.
\s	A whitespace character: *[\t\n\x0B\f\r]*	This matches tab, space, end-of-line, carriage-return, form-feed, and newline characters.
\S	A non-whitespace character	Matches anything that is not a whitespace character, as described previously. Thus, *7* would be matched, as would punctuation.
\w	A word character: *[a-zA-Z_0-9]*	Any uppercase or lowercase letter, digit, or the underscore character.
\W	A nonword character; the opposite of *\w*	Anything that isn't a word character, as described previously. Thus, the minus sign will match, as will a space. It won't match the end-of-line *$* or beginning-of-line ^ characters.

Table A-3. Character Classes

Regex	Description	Notes
[abc]	*a*, *b*, or *c*	Strictly speaking, it won't match *ab*.
[^abc]	Any character except *a*, *b*, or *c*	This will match any character except *a*, *b*, or *c*. It won't match the end-of-line ***$*** or beginning-of-line ^ characters.
[a-zA-Z]	Any uppercase or lowercase letter	When working with numbers, ***[0-25]*** doesn't mean 0 to 25. It means 0 to 2, or just 5. If you wanted 0 to 25, you would need to actually write an expression, such as ***\d|1\d|2[0-5]***. Note that ***[0-9]*** is exactly equal to ***\d***.
[a-c[x-z]]	*a* through *c*, or *x* through *z*	For example, ***[1-3[7-9]]*** matches 1 through 3, or 7 through 9. No other digit will do.
[a-z&&[a,e]]	*a* or *e*	***[a-z&&[a,e,i,o,u]]*** matches all lowercase vowels.
[a-z&&[^bc]]	All lowercase letters except for *b* and *c*	For example, all the prime numbers between 1 and 9 would be ***[1-9&&[^4689]]***. That is, 1 through 9, excluding 4, 6, 8, and 9.
[a-d&&[^b-c]]	*a* through *d*, but not *b* through *c*	***[1-9&&[^4-6]]*** matches 1 through 3, or 7 through 9. Compare this to the union example presented earlier in this table.

Table A-4. POSIX Character Classes

Regex	Description	Notes
\p{Lower}	A lowercase alphabetic character	
\p{Upper}	An uppercase alphabetic character	
\p{ASCII}	An ASCII character	
\p{Alpha}	An alphabetic character	
\p{Digit}	A decimal digit	
\p{Alnum}	An alphanumeric character	
\p{Punct}	Punctuation	This is a good way to deal with punctuation in general, without having to delimit special characters such as periods, parentheses, brackets, and such. It matches *!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~*.
\p{Graph}	A visible character	Exactly equal to ***[\p{Alnum}\p{Punct}]***.
\p{Print}	A printable character	Exactly equal to ***[\p{Graph}]***.
\p{Blank}	A space or a tab	
\p{Space}	Any whitespace character	It matches ***[\t\n\x0B\f\r]***.
\p{Cntrl}	A control character	
\p{XDigit}	A hexadecimal digit	

Table A-5. Boundary Matchers

Regex	Description	Notes
^	Beginning-of-line character	This is an invisible character.
$	End-of-line character	This is an invisible character.
\b	A word boundary	This is the position of a word boundary. Its usage requires some caution, because ***\b*** doesn't match characters; it matches a position. Thus, the `String` *anna marrie* doesn't match the regex ***anna\bmarie***. However, it does match ***anna\b\smarrie***. That's because there's a word boundary at the position after the last ***a*** in *anna*, and it happens to be the space character, so ***\s*** is necessary to match it, and *marie* must then follow it. Because ***\b*** matches a position, it is meaningless to add greedy qualifiers to it. Thus, ***\b+***, ***\b\b\b\b\b***, and ***\b*** all match exactly the same thing. Further complicating the picture is the fact that in a character class, ***\b*** means a backspace character. This is syntactically legal (if a little awkward) because the word boundary has no place inside a character class. Thus, *[\b]* describes a backspace character, because it is surrounded by *[* and *]*.
\B	A non-word boundary	This is the opposite of word boundary, as described previously.
\A	The beginning of the input	***\A*** matches the beginning of the input, but it isn't just a synonym for the ^ pattern. This distinction becomes clear if you use the `Pattern.MULTILINE` flag when you compile your pattern. ***\A*** matches the beginning of the input, which is the very beginning of the file. By contrast, ^ matches the beginning of each line when the `Pattern.MULTILINE` flag is active.
\Z	The end of the input except for the final $, if any	***\Z*** matches the end of the input, but it isn't just a synonym for the ***$*** character. This distinction becomes clear if you use the `Pattern.MULTILINE` flag when you compile your pattern. ***\Z*** matches the end of the input, which is the very end of the file. By contrast, ***$*** matches the end of each line when the `Pattern.MULTILINE` flag is active.
\G	The end of the previous match	
\z	The end of the input	This behaves exactly like the ***\Z*** with a capital *Z* character, except that it also captures the closing ***$*** character.

Table A-6. Greedy Quantifiers

Regex	Description	Notes
X?	*X*, once or not at all	*A?* would match *A*, or the absence of *A*. This applies to either the character that immediately precedes it, or a group (if the group immediately precedes it), or a character class (if the character class immediately precedes it).
X*	*X*, zero or more times	This pattern is very much like the *?* pattern, except that it matches zero or more occurrences. It doesn't match "any character," as its usage in DOS might indicate. Thus, *A** would match *A*, *AA*, *AAA*, or the absence of *A*. This applies to either the character that immediately precedes it, or a group (if the group immediately precedes it), or a character class (if the character class immediately precedes it).
X+	*X*, one or more times	This quantifier is very much like the * pattern, except that it looks for the existence of one or more occurrences instead of zero or more occurrences. This applies to either the character that immediately precedes it, or a group (if the group immediately precedes it), or a character class (if the character class immediately precedes it).
X{n}	*X*, exactly *n* times	This quantifier demands the occurrence of the target exactly *n* times. This applies to either the character that immediately precedes it, or a group (if the group immediately precedes it), or a character class (if the character class immediately precedes it).
X{n,}	*X*, at least *n* times	This quantifier demands the occurrence of the target at least *n* times. This applies to either the character that immediately precedes it, or a group (if the group immediately precedes it), or a character class (if the character class immediately precedes it).
X{n,m}	*X*, at least *n* but not more than *m* times	This quantifier demands the occurrence of the target at least *n* times, but not more than *m* times. This applies to either the character that immediately precedes it, or a group (if the group immediately precedes it), or a character class (if the character class immediately precedes it).

Table A-7. Reluctant Quantifiers

Regex	Description	Notes
X??	*X*, once or not at all	This pattern is very much like the *?* pattern, except that it prefers to match nothing at all. When it's used with the `Matcher.matches()` method, *??* functions in exactly the same way as the *?* pattern. However, when it's used with `Matcher.find()`, the behavior is different. For example, the pattern *x??*, as applied to the `String` *xx*, will actually not find *x*, yet consider that lack of finding a success. That's because we asked it to be reluctant to match, and the most reluctant thing it can do is match zero occurrences of *x*. This applies to either the character that immediately precedes it, or a group (if the group immediately precedes it), or a character class (if the character class immediately precedes it).
X*?	*X*, zero or more times	This pattern is very much like the * pattern, except that it prefers to match as little as possible. When it's used with the `Matcher.matches()` method, **?* functions in exactly the same way as the * pattern. However, when it's used with `Matcher.find()`, the behavior is different. For example, the pattern *x*?*, as applied to the `String` *xx*, will actually not find *x*, yet consider that a success. That's because we asked it to be reluctant to match, and the most reluctant thing it can do is match zero occurrences of *x*. This applies to either the character that immediately precedes it, or a group (if the group immediately precedes it), or a character class (if character the class immediately precedes it).
X+?	*X*, one or more times	This pattern is very much like the + pattern, except that it prefers to match as little as possible. When it's used with the `Matcher.matches()` method, *+?* functions in exactly the same way as the + pattern. However, when it's used with `Matcher.find()`, the behavior is different. For example, the pattern *x+?*, as applied to the `String` *xx*, will actually find one *x*, yet consider that a success. That's because we asked it to be reluctant to match, and the most reluctant thing it can do is match one occurrence of *x*. This applies to either the character that immediately precedes it, or a group (if the group immediately precedes it), or a character class (if the character class immediately precedes it).

Table A-7. Reluctant Quantifiers (Continued)

Regex	Description	Notes
X{n}?	*X*, exactly *n* times	This pattern is exactly like the ***X{n}*** pattern. This applies to either the character that immediately precedes it, or a group (if the group immediately precedes it), or a character class (if the character class immediately precedes it).
X(n,}?	*X*, at least *n* times	This pattern is very much like the ***X{n,}*** pattern, except that it prefers to match as little as possible. When it's used with the `Matcher.matches()` method, ***X{n,}?*** functions in exactly the same way as the ***X{n,}*** pattern. However, when it's used with `Matcher.find()`, the behavior is different. For example, the pattern ***X{3,}?***, as applied to the `String` *xxxxx*, will actually only find *xxx*, yet consider that a success. Compare this with just ***X{3,5}***, which would have found *xxxxx*. That's because we asked it to be reluctant to match, and the most reluctant thing it can do is match three occurrences of *x*. This applies to either the character that immediately precedes it, or a group (if the group immediately precedes it), or a character class (if the character class immediately precedes it).
X{n,m}?	*X*, at least *n* but not more than *m* times	This pattern is very much like the ***X{n,m}*** pattern, except that it prefers to match as little as possible. When it's used with the `Matcher.matches()` method, ***X{n,m}?*** functions in exactly the same way as the ***X{n,m}*** pattern. However, when it's used with `Matcher.find()`, the behavior is distinct. For example, the pattern ***X{3,5}?***, as applied to the `String` *xxxxx*, will actually find *xxx*, yet consider that a success. Compare this with just ***X{3,5}***, which would have found *xxxxx*. This happens because we asked it to be reluctant to match, and the most reluctant thing it can do is match three occurrences of *x*. Notice that if there were six *x* characters, such as *xxxxxx*, the pattern would have matched twice: once for the first three *xxx* characters and then again for the other three. Again, this is because three is the minimum requirement. This applies to either the character that immediately precedes it, or a group (if the group immediately precedes it), or a character class (if the character class immediately precedes it).

Table A-8. Possessive Quantifiers

Regex	Description	Notes
X?+	*X*, once or not at all	Very much like the **?** pattern, this pattern prefers to match as much as possible. However, this pattern won't release matches to help the entire expression match as a whole. For example, the pattern ***\w?+\d***, as applied to the `String` *A2*, will actually not match, because the first ***\w?+*** consumes the *A* and the *2*, and it won't release them for the greater good of allowing the entire expression to match. Thus, ***\d*** is unable to match. This is because we asked ***\w?+*** to be possessive, and the most possessive thing it can do is match the occurrence of *A* and *2*, and not release them. This applies to either the character that immediately precedes it, or a group (if the group immediately precedes it), or a character class (if the character class immediately precedes it).
X*+	*X*, zero or more times	Very much like the * pattern, this pattern prefers to match as much as possible. However, this pattern won't release matching to help the entire expression as a whole match. For example, the pattern ***\w*+\d***, as applied to the `String` *Java2*, will actually not match, because the first ***\w*+*** consumes the `String` *Java2* and won't release it for the greater good of allowing the entire expression to match. Thus, ***\d*** is unable to match. This is because the pattern ***\w*+*** is possessive, and the most possessive thing it can do is match the entire *Java2* and not release anything. This applies to either the character that immediately precedes it, or a group (if the group immediately precedes it), or a character class (if the character class immediately precedes it).

Table A-8. Possessive Quantifiers (Continued)

Regex	Description	Notes
X++	*X*, one or more times	Very much like the + pattern, this pattern prefers to match as much as possible. However, this pattern won't release matching to help the entire expression as a whole match. For example, the pattern *\w++\d*, as applied to the `String` *Java2*, will actually not match, because the first *\w++* consumes the `String` *Java2* and won't release it for the greater good of allowing the entire expression to match. Thus, *\d* is unable to match. This is because the pattern *\w++* is possessive, and the most possessive thing it can do is match the entire *Java2* and not release anything. This applies to either the character that immediately precedes it, or a group (if the group immediately precedes it), or a character class (if the character class immediately precedes it).
X{n}+	*X*, exactly *n* times	This pattern is exactly like the *X{n}* pattern. This applies to either the character that immediately precedes it, or a group (if the group immediately precedes it), or a character class (if the character class immediately precedes it).
X{n,}+	*X*, at least *n* times	Very much like the *X{n,}* pattern, this pattern prefers to match as much as possible. However, this pattern won't release matching to help the entire expression as a whole match. For example, the pattern *\w{4,}+\d*, as applied to the `String` *Java2*, will actually not match, because the *\w{4,}+* consumes the `String` *Java2* and won't release it for the greater good of allowing the entire expression to match. Thus, *\d* is unable to match *4*. This is because the pattern *\w{4,}+* is possessive, and the most possessive thing it can do is match the entire *Java2* and not release anything. This pattern applies to either the character that immediately precedes it, or a group (if the group immediately precedes it), or a character class (if the character class immediately precedes it).

Table A-8. Possessive Quantifiers (Continued)

Regex	Description	Notes
X{n,m}+	*X*, at least *n* but not more than *m* times	Very much like the ***X{n,m}*** pattern, this pattern prefers to match as much as possible. However, this pattern won't release matching to help the entire expression as a whole match. For example, the pattern ***\x{2,5}\d***, as applied to the `String` *Java2*, will actually not match, because the first ***\w++*** consumes the `String` *Java2*, and won't release it for the greater good of allowing the entire expression to match. Thus, ***\d*** is unable to match. This is because the pattern ***\x{2,5}*** is possessive, and the most possessive thing it can do is match the entire *Java2* and not release anything. This applies to either the character that immediately precedes it, or a group (if the group immediately precedes it), or a character class (if the character class immediately precedes it).

Table A-9. Logical Operators

Regex	Description	Notes
XY	*X* followed by *Y*	This is the default relationship assumption between characters. Note that spaces are a valid part of this syntax. Thus, ***A B*** means the character *A*, followed by a space, followed by the character *B*.
X|Y	Either *X* or *Y*	***AB|CD*** will match either *AB* or *CD*. Similarly, the pattern ***hello sir|madam*** will match *hello sir* or it will match *madam*. Specifically, it won't match *hello madam*. This is because of the nature of the And pattern discussed previously. When the regex engine sees *hello sir*, it assumes you mean that *hello*, followed by a space, followed by *sir* should be treated as a single logical unit. These are all Anded together. Then the engine sees the Or pattern, so it assumes that the logical alternative is *madam*. If you actually want to accept *hello sir* or *hello madam*, you'll have to use groups—thus, the pattern ***hello (sir|madam)***. Or better yet, you can use the noncapturing group ***hello (?:sir|madam)***.

Table A-9. Logical Operators (Continued)

Regex	Description	Notes
(X)	*X*, the capturing group	A capturing group is a logical unit that is conceptually similar to the logical units you're familiar with from algebra. Thus, *(\d\d\d\d)* is a capturing group that defines four digits. Capturing groups can be referred to later in your expression by using a back reference, as explained later. They're counted left to right and can be nested. Thus, *h(ello (world))* has three capturing groups. Capturing group 0 is the entire expression, which matches the `String` *hello world*. Capturing group 1 is *ello world*, because you count from left to right, and the first group starts with the *(* right before the *e* in *hello*. Group 2 is *world*, because the second group starts right before the *w* in *world*.
\n	The *n*th capturing group matched	In this context, I'm not referring to newline, even though *\n* looks like the newline symbol. The *n* in this case refers to a number. The regex engine allows you to access the information captured by a previous part of the group, even as the search is executing. For example, if you want to find repeated words, all you need is the pattern *(\w+)\W\1*, which says, "Look for a group of word characters, followed by a nonword character, followed by that exact same word character captured in group 1." If you attempt to refer to a group that doesn't exist, a `PatternSyntaxException` will be thrown. If you happen to have, say, 13 captured groups, then *\13* will mean that you want the thirteenth capturing group. If you don't have 13 groups, then the same expression *\13* will mean the first capturing group, followed by the digit *3*.

Table A-10. Quotation

Regex	**Description**	**Notes**
\	Quotes the following character	This quotes the metacharacter that follows, so it will actually be treated as a character. Thus, if you were looking for a dollar sign, you would use *\$*. as the pattern. By contrast, *$* would have matched the end-of-line character. Remember that for regex expressions used directly as `Strings`, you need to double the number of \ characters you see. Thus, in a Java `String`, *\s* becomes *\\s*.
\Q	Quotes all characters until *\E*	This works in conjunction with *\E* to quote a sequence of characters. If you need to quote a lot of characters in sequence, then use *\Q* to open your quote and *\E* to close it. For example, if you want the characters *\([?**, the expression *\Q\(?*\E* will do the job.
\E	Ends quote started by *\Q*	

Table A-11. Noncapturing Group Constructs

Regex	**Description**	**Notes**
(?:X)	Defines a subpattern as a logical unit	Noncapturing groups don't store the information that actually matches the pattern for later access. These are much more efficient than capturing groups if you're only using grouping for logical purposes. This pattern is noncapturing.
(?idmsux-idmsux)	*i* for `CASE_INSENSITIVE` *x* for `COMMENTS` *s* for `DOTALL` *u* for `UNICODE_CASE` *m* for `MULTILINE` *d* for `UNIX_LINES`	The pattern *(?i)hel(?-i)LO* will match the `String` *HELLO*, because *(?i)* indicates a case-insensitive match starting from ***h***, and *(?-i)* signals an end to that case insensitivity after the first ***l***. This pattern is noncapturing.
(?idmsux-idmsux:X)	*X*, with the given flags on or off	The pattern *(?i:hel)LO* will match the `String` *HELLO*, because *(?i:* indicates a case-insensitive match starting from ***h*** and ending with the first ***l***. This pattern is noncapturing.

Table A-12. Lookarounds

Regex	Description	Notes
(?=X)	*X*, using zero-width positive lookahead	This pattern glances to the right of whatever remains to be parsed from the candidate `String` to find the first position at which the expression *X* exists. For example, if you want to extract all of the inline comments from a text file, you might try the pattern *(?=//).*$* and extract group 0. This pattern is noncapturing.
(?!X)	*X*, using zero-width negative lookahead	This pattern glances to the right, to whatever remains to be parsed from the candidate `String`, to find the first position at which the expression *X* doesn't exist. For example, if you want to skip leading spaces leading up to some content, you could use *(?!\s).**. This pattern is noncapturing.
(?<=X)	*X*, using zero-width positive lookbehind	This pattern glances to the left, to whatever remains to be parsed from the candidate `String`, to find the first position at which the expression *X* exists. For example, if you want to extract all of the inline comments from a text file, you might try the pattern *(?=//).*$*. This pattern is noncapturing.
(?<!X)	*X*, using zero-width negative lookbehind	This pattern glances to the left, to whatever remains to be parsed from the candidate `String`, to find the first position at which the expression *X* doesn't occur. For example, if you want to extract all of the text before inline Java comments from a text file, you might try the pattern *.*(?<=//)*. This pattern is noncapturing.
(?>X)	*X*, as an independent, non-capturing group	This pattern refuses to release the contents of the match, regardless of the consequences on the rest of the pattern's ability to match. Thus, whereas the pattern *\w+\d* matches the `String` *java2*, the pattern *(?>\w+)\d* does not, because the *(?>\w+)* consumes *java* and *2*, and refuses to release the *2* so that *\d* can match.

Table A-13. Less Common Characters

Regex	Description	Notes
\0n	The character with octal value *on*	0 <= n <= 7
\0nn	The character with octal value *onn*	0 <= n <= 7
\0mnn	The character with octal value *0mnn*	0 <= m <= 3, 0 <= n <= 7 (This can't exceed 377.)
\xhh	The character with hexadecimal value *0xhh*	0 <= h <= 9 or A<=h <=F
\uhhhh	The character with hexadecimal value *0xhhhh*	0 <= h <= 9 or A<=h <=F
\a	The alert (bell) character (`'\u0007'`)	
\e	The escape character (`'\u001B'`)	
\cx	The control character corresponding to *x*	

Table A-14. Unicode Blocks and Categories

Regex	Description	Notes
\p{InGreek}	A character in the Greek block	
\p{Lu}	An uppercase letter	***\p{Lu}*** matches any uppercase character.
\p{Sc}	A currency symbol	If you need to find or swap out, say, a dollar sign, this is a good way to do so without having to deal with the various delimiting complexities of not matching the end-of-line character ***$***.
\P{InGreek}	Any character except one in the Greek block (negation)	Notice the use of the capital *P* here. In general, uppercase ***\P*** is the opposite of lowercase ***\p***. Thus, ***\P{Lower}*** matches all uppercase characters.
[\p{L}&&[^\p{Lu}]]	Any non-uppercase letter	This is exactly equal to ***\p{Upper}***.

APPENDIX B

Pattern and Matcher Methods

THIS APPENDIX PROVIDES a summary of the methods of the `Pattern` and `Matcher` classes in Java. It's intended to be a quick reference for working with the various regex utilities you'll be using. For more detailed descriptions, please see the appropriate section in the text.

Pattern Class Fields

UNIX_LINES

The `UNIX_LINES` flag is used in constructing the second parameter of the `Pattern.compile(String regex, int flags)` method. Use this flag when parsing data that originates on a UNIX machine.

On many flavors of UNIX, the invisible character *\n* is used to note termination of a line. This is distinct from other operating systems, including flavors of Windows, which may use *\r\n*, *\n*,*\r*, *\u2028*, or *\u0085* for a line terminator.

If you've ever transported a file that originated on a UNIX machine to a Windows platform and opened it, you may have noticed that the lines sometimes don't terminate as you might expect, depending on which editor you use to view the text. This happens because the two systems can use different syntax to denote the end of the line.

The `UNIX_LINES` flag simply tells the regex engine that it's dealing with UNIX style lines, which affects the matching behavior of the regular expression metacharacters ^ and *$*.

> **NOTE** Using the `UNIX_LINES` flag, or the equivalent *(?d)* regex pattern, doesn't degrade performance. By default, this flag isn't set.

CASE_INSENSITIVE

The `CASE_INSENSITIVE` field is used in constructing the second parameter of the `Pattern.compile(String regex, int flags)` method. It's useful when you need to match U.S. ASCII characters, regardless of case.

> **NOTE** Using this flag, or the equivalent *(?i)* regular expression, can cause performance to degrade slightly. By default, this flag is not set.

COMMENTS

The `COMMENTS` field is defined because it's used in constructing the second parameter of the `Pattern.compile(String regex, int flags)` method. It tells the regex engine that the regex pattern has an embedded comment in it. Specially, it tells the regex engine to ignore any comments in the pattern, starting with the spaces leading up to the # character and everything thereafter, until the end of the line.

Thus, the regex pattern ***A #matches uppercase US-ASCII char code 65*** will use *A* as the regular expression, but the spaces leading up to the # character and everything thereafter until the end of the line will be ignored.

> **NOTE** Using this flag, or the equivalent *(?x)* regular expression, doesn't degrade performance.

MULTILINE

The `MULTILINE` field is used in constructing the second parameter of the `Pattern.compile(String regex, int flags)` method. It tells the regex engine that regex input isn't a single line of code; rather, it contains several lines that have their own termination characters.

This means that the beginning-of-line character, ^, and the end-of-line character, $, will potentially match several lines within the input `String`.

For example, imagine that your input `String` is *This is sentence.\n So is this.* If you use the MULTILINE flag to compile the regular expression pattern:

```
Pattern p = Pattern.compile("^", Pattern.MULTILINE);
```

then the beginning of line character, ^, will match before the *T* in *This is a sentence.* It will also match just before the *S* in *So is this.* Without using the `MULTILINE` flag, the match will only find the *T* in *This is a sentence.*

> **NOTE** Using this flag, or the equivalent *(?m)* regular expression, may degrade performance.

DOTALL

The `DOTALL` flag is used in constructing the second parameter of the `Pattern.compile(String regex, int flags)` method.

The `DOTALL` flag tells the regex engine to allow the metacharacter period (.) to match any character, *including* a line termination character. What does this mean?

Imagine that your candidate `String` were *Test\n*. If your corresponding regex pattern were the period (.), then you would normally have four matches: one for the *T*, another for the *e*, another for *s*, and the fourth for *t*. This is because the regex metacharacter period (.) will normally match any character, except line termination characters.

Enabling the `DOTALL` flag

```
Pattern p = Pattern.compile(".", Pattern.DOTALL);
```

would have generated five matches. Your pattern would have matched the *T*, *e*, *s*, and *t* characters. In addition, it would have matched the *\n* character at the end of the line.

> **NOTE** Using this flag, or the equivalent *(?s)* regular expression, doesn't degrade performance.

UNICODE_CASE

The `UNICODE_CASE` flag in conjunction with the `CASE_INSENSITIVE` flag generates case-insensitive matches for international character sets.

> **NOTE** Using this flag, or the equivalent *(?u)* regular expression, can degrade performance.

CANON_EQ

As you know, characters are actually stored as numbers. For example, in the U.S. ASCII character set, the character A is represented by the number 65. Depending on the character set that you're using, the same character can be represented by different numeric combinations. For example, *à* can be represented by both +00E0 and U+0061U+0300. A `CANON_EQ` match would match either representation.

> **NOTE** Using this flag may degrade performance.

Pattern Class Methods

public static Pattern compile(String regex) throws PatternSyntaxException

You'll notice that the `Pattern` class doesn't have a public constructor. This means that you can't write the following type of code:

```
Pattern p = new Pattern("my regex");//wrong!
```

To get a reference to a `Pattern` object, you must use the static method `pattern(String regex)`. Thus, your first line of regex code might look like the following:

```
Pattern p = Pattern.compile("my regex");//Right!
```

The parameter for this method is a `String` that represents a regular expression. When passing a `String` to a method that expects a regular expression, it's important to delimit any \ characters that the regular expressions might have by appending another \ character to them. This is because `String` objects internally use the \ character to delimit metacharacters in a character sequences, regardless of whether those character sequences are regular expressions. This has been true long before regular expression were a part of Java. Thus, the regular expression ***\d*** becomes ***\\d***. To match a single digit, your regular expression code becomes the following:

```
Pattern p = Pattern.compile("\\d");
```

The point here is that the regular expression ***\d*** becomes the `String` ***\\d***.

The delimitation of the `String` parameter can sometimes be tricky, so it's important to understand it well. By and large, it means that you double the \ characters that might already be present in the regular expression. It doesn't mean that you simply append a single \ character.

The compile method will throw a java.util.regex.PatternSyntaxException if the regular expression itself is badly formed. For example, if you pass in a `String` that contains ***[4***, the compile method will throw a `PatternSyntaxException` at runtime, because the syntax of the regular expression ***[4*** is illegal.

The `compile(String regex)` method returns a `Pattern` object.

public static compile pattern(String regex, int flags) throws PatternSyntaxException

The `compile(String regex, int flags)` method is a more powerful form of the `compile` method. The first parameter for this method, `regex`, is a `String` that represents a regular expression, as detailed in the previous `pattern.compile(String regex)` method entry. For details on how the `String` parameter must be formatted, please see the previous `compile(String regex)` method entry.

The flexibility of this `compile` method is fully realized by using the second parameter, `int flags`. For example, if you want a match to be successful regardless of the case of the candidate `String`, then your pattern might look like the following:

```
Pattern p = Pattern.compile(regex,Pattern.CASE_INSENSITIVE);
```

You can combine the flags by using the | operator. For example, to achieve case-insensitive Unicode matches that include a comment, you might use the following:

```
Pattern p =
Pattern.compile("t # a compound flag example",Pattern.CASE_INSENSITIVE |
        Pattern.UNICODE_CASE| Pattern.COMMENT);
```

The `compile(String regex, int flags)` method returns a `Pattern` object.

public String pattern()

This method returns the regular expression from which this pattern was compiled. This is a simple `String` that represents the regex you passed in.

This method can be misleading in two ways. First, the `String` that is returned doesn't reflect any flags that were set when the pattern was compiled. Second, the regex `String` you passed in isn't always the pattern `String` you get back out. Specifically, the original `String` delimitations aren't shown. Thus, if your original code was

```
Pattern p = Pattern.compile("\\d");
```

then you should expect your output to be ***\d***, with a single \ character.

public Matcher matcher(CharSequence input)

Remember that you create a `Pattern` object by compiling a description of what you're looking for. A `Pattern` lists the features of what you're looking for. Speaking purely conceptually, your patterns might look like the following:

```
Pattern p = Pattern.compile("She must have red hair and be smarter than I am");
```

Correspondingly, you'll need to compare that description against candidates. That is, you'll want to examine a given `String` to see whether it matches the description you provided.

The `Matcher` object is designed specifically to help you do this sort of interrogation. The `Pattern.matcher(CharSequence input)` method returns the `Matcher` that will help get details about how your candidate `String` compares with the description you passed in.

public int flags()

Earlier, I discussed the constant flags that you can use in compiling your regex pattern. The flags method simply returns an `int` that represents those flags.

To see if your Pattern class is currently using a given flag—for example, the Pattern.COMMENTS flag—simply extract the flag

```
int flgs  = myPattern.flags();
```

and then & that flag to the Pattern.COMMENTS flag:

```
boolean isUsingCommentFlag =(  Pattern.COMMENTS == (Pattern.COMMENTS & flgs)) ;
```

Similarly, to see if you're using CASE_insensitive, do the following:

```
boolean isUsingCaseInsensitiveFlag =
(Pattern.CASE_insensitive == (Pattern. CASE_insensitive & flgs));
```

public static boolean matches(String regex,CharSequence input)

Very often, you'll find that all you need to know about a String is whether it matches a given regular expression exactly. You don't want to have to create a Pattern object, extract its Matcher object, and interrogate that Matcher.

This static utility method is designed to help you do exactly that. Internally, it creates the Pattern and Matcher objects you need, compares the regex to the input String, and returns a Boolean indicating whether the two objects match exactly. Usage might look something like PatternMatchesTest example shown here:

```
import java.util.regex.*;
public class PatternMatchesTest{
  public static void main(String args[]){

      String regex = "ad*";
      String input = "add";

      boolean isMatch = Pattern.matches(regex,input);
      System.out.println(isMatch);\\return true
  }
}
```

If you're going to be doing a lot of comparisons, then it's more efficient to explicitly create a Pattern object and do your matches manually. However, if you're not going to be doing a lot of comparisons, then matches is a handy utility method.

The `Pattern.matches(String regex, CharSequence input)` method is also used internally by the `String` class. As of J2SE 1.4, `String` has a new method called `matches`, which internally defers to this one. Thus, you might already be using this method without being aware of it.

Of course, this method can throw a `PatternSyntaxException` if the regex pattern under consideration isn't well formed.

public String[] split(CharSequence input)

This method can be particularly helpful if you need to break up a `String` into an array of substrings based on some criteria. In concept, it's similar to the `StringTokenizer`. However, it's much more powerful and more resource intensive than `StringTokenizer`, because it allows your program to use a regular expressions as the splitting criteria.

This method always returns at least one element. If the split candidate, `input`, can't be found, a `String` array is returned that contains exactly one `String`, namely the original `input`.

If the `input` can be found, then a `String` array is returned. That array contains every substring after an occurrence of the `input`. Thus, for the pattern

```
Pattern p = new Pattern.compile(",");
```

the split method for *Hello, Dolly* will return a `String` array consisting of two elements. The first element of the array will contain the `String` *Hello*, and the second will contain the `String` *Dolly*. That `String` array is obtained as follows:

```
String tmp[] = p.split("Hello,Dolly");
```

In this case, the value return is

```
//tmp ={ "Hello", "Dolly"}
```

There are some subtleties you should be aware of when working with this method. If the candidate `String` had been *Hello,Dolly,* with a trailing comma character after the *y* in *Dolly,* then this method would still have returned two elements: a `String` array consisting of *Hello* and *Dolly*. The implicit behavior is that training spaces aren't returned.

If the input `String` had been *Hello,,,Dolly,* the resulting `String` array would have four elements. The return value of the split method, as applied to the `Pattern` , is

```
// p.split("Hello,,,Dolly") == {"Hello","","","Dolly"}
```

The `String` method further optimizes its search criteria by placing an invisible ^ before the pattern and a *$* after it.

public String[] split(CharSequence input, int limit)

This method works in exactly the same way as `Pattern.split(CharSequence input)`, with one variation. The second parameter, `limit`, allows you to control how many elements are returned:

```
Limit == 0
```

If you specify that the second parameter, `limit`, should equal 0, then this method behaves exactly like its overloaded counterpart:

```
Limit >0
```

Use a positive limit if you're interested in only a certain number of matches. You should use number *1* as the limit. Say the `Pattern` **p** has been compiled for the `String` , as previously. To split the `String` *Hello, Dolly, You, Are, My, Favorite* when you only want the first two tokens, you would use this:

```
String[] tmp = pattern.split("Hello, Dolly, You, Are, My, Favorite",3);
```

The value of the resulting `String` would be this:

```
//tmp[0] =  "Hello",  tmp[1] = "Dolly";
```

The interesting behavior here is that a third element is returned, in this case

```
//tmp[2] =  "You, Are, My, Favorite";
```

Using a positive limit can potentially lead to performance enhancements, because the regex engine can stop searching when it meets the specified number of matches:

```
Limit <0
```

Using a negative number—any negative number—for the limit tells the regex engine that you want to return as many matches as possible *and* that you want trailing spaces, if any, to be returned. Thus, for the regex pattern , and the candidate `String` *Hello,Dolly* the command

```
String tmp[] = p.split("Hello,Dolly", -1);
```

results in

```
//tmp == {"Hello","Dolly"};
```

However, for the String *Hello, Dolly,<space><space><space>*, with trailing spaces after the comma following the *Dolly,* the method call

```
String tmp[] = p.split("Hello,Dolly,   ", -1);
```

results in

```
//tmp == {"Hello","Dolly","   "};
```

Notice that the actual value of the negative limit doesn't matter. Thus,

```
p.split("Hello,Dolly", -1);
```

is exactly equivalent to

```
p.split("Hello,Dolly", -100);
```

Method Class Methods

public Pattern pattern()

The pattern method returns the Pattern that created this particular Matcher object. The Pattern returned doesn't contain any of the flags that are explicitly set by using the Pattern constants when the pattern is compiled, such as Pattern.MULTILINE.

public Matcher reset()

The reset() method clears all state information from the Matcher object it is called on. The Matcher is, in effect, reverted to the state it originally had when you first received a reference to it.

public Matcher reset(CharSequence input)

The reset(CharSequnce input) method clears the state of the Matcher object it's called on and replaces the candidate String with the new input. This has the same effect as creating a new Matcher object, except that it doesn't have the associated overhead. This recycling can be a useful optimization, and it's one that I often use.

public int start()

This method returns the index of the first character of the candidate `String` matched. If there are no matches, or if no matches have been attempted, an `IllegalStateException` is thrown.

public int start(int group)

This method allows you to specify which subgroup within a matching you're interested in and returns the index of the first character in which that subgroup starts. If there are no matches, or if no matches have been attempted, an `IllegalStateException` is thrown. If you refer to a group number that doesn't exist, an `IndexOutOfBoundsException` is thrown.

public int end()

The `end` method returns the ending index plus 1 of the last successful match the `Matcher` object had. If no matches exist, or if no matches have been attempted, this method throws an `IllegalStateException`.

public int end(int group)

Like the `start(int)` method, this method allows you to specify which subgroup within a matching you're interested in, except that it returns the last index of matching character sequence plus 1. If no matches exist, or if no matches have been attempted, this method throws an `IllegalStateException`. If you refer to a group number that doesn't exist, an `IndexOutOfBoundsException` is thrown.

public String group()

The `group` method can be a powerful and convenient tool in the war against jumbled code. It simply returns the substring of the candidate `String` that matches the original regex pattern. The `group()` method throws an `IllegalStateException` if the `find` method call wasn't successful. Similarly, it throws an `IllegalStateException` if `find` has never been called at all.

public String group(int group)

This method is a more powerful counterpart to the `group()` method. It allows you to extract parts of a candidate string that match a subgroup within your pattern.

The `group(int)` method throws an `IllegalStateException` if the `find` method call wasn't successful. Similarly, it throws an `IllegalStateException` is `find` has never been called at all. If called for a group number that doesn't exist, it throws an `IndexOutOfBoundsException`.

public int groupCount()

This method simply returns the number of groups that the `Pattern` defined. There's a very important, and somewhat counterintuitive, subtlety to notice about this method: It returns the number of possible groups based on the original `Pattern`, without even considering the candidate `String`. Thus, it's not really information about the `Matcher` object; instead, it's about the `Pattern` that helped spawn it. This can be tricky, because the fact that this method lives on the `Matcher` object could be interpreted as meaning that it's providing feedback about the state of the `Matcher`. It isn't. It's telling you how many matches are theoretically possible for the given `Pattern`.

public boolean matches()

This method is designed to help you match a candidate `String` against the matcher's `Pattern`. It returns true if, and only if, the candidate `String` under consideration matches the pattern exactly.

public boolean find()

The `find()` method parses just enough of the candidate `String` to find a match. If such a substring is found, then true is returned and `find` stops parsing the candidate. If no part of the candidate `String` matches the pattern, then `find` returns false.

public boolean find(int start)

`find(int start)` works exactly like its overloaded counterpart, with the exception of where it starts searching. The `int` parameter `start` simply tells the `Matcher` at which character to start its search.

Thus, for the candidate `String` *I love Java. Java is my favorite language. Java Java Java.* and the `Pattern` **Java**, if you only want to start searching at character index 11, you use the command `find(11)`.

public Matcher appendReplacement(StringBuffer sb, String replacement)

The appendReplace method allows you to modify the contents of a StringBuffer based on a regular expression. It even allows you to use back references by using the ***$n*** notation, in which ***n*** refers to some captured subgroup. For example, for the StringBuffer *Waldo Smith*, the Pattern *(\w+) (\w+)*, and the replacement ***$2, $1***, the contents of the StringBuffer will be modified to *Smith, Waldo.*

The appendReplacement method will throw an IllegalStateException if a find() hasn't been called, or if find would have returned false if called. It will throw an IndexOutOfBoundsException if the capturing group referred to by ***$1, $2***, etc., doesn't exist in the part of the pattern currently being scrutinized by the Matcher.

public StringBuffer appendTail(StringBuffer sb)

The appendTail method is a supplement to the appendReplacement method. It simply appends every remaining subsequence from the original candidate String to the StringBuffer, if reading from the append position, which I explained in the appendReplacement section, to the end of the candidate string.

public String replaceAll(String replacement)

The replaceAll method returns a String that replaces every occurrence of the description with the replacement. Using this method will change the state of your Matcher object. Specifically, the reset() method will be called. Therefore, remember that all start, end, group, and find calls will have to be reexecuted.

Like the appendReplacement method, the replaceAll method can contain references to substring by using the ***$*** symbol. For details, please see the appendReplacement documentation presented earlier.

public String replaceFirst(String replacement)

The replaceFirst method is a more limited version of the replaceAll method. This method returns a String that replaces the first occurrence of the description with the replacement. Using this method will change the state of your Matcher object. Specifically, the reset() method will be called. Therefore, remember that all start, end, group, and find calls will have to be reexecuted after replaceFirst is called.

Like the appendReplacement method, the replaceFirst method can contain references to substring by using the ***$*** symbol. For details, please see the appendReplacement documentation presented earlier.

APPENDIX C

Common Regex Patterns

THIS APPENDIX PRESENTS some practical regex patterns that you can use for common matching and validation tasks.

Table C-1. IP Address ^(([0-1]?\d{1,2}\.)|(2[0-4]\d\.)|(25[0-5]\.)){3} (([0-1]?\d{1,2})|(2[0-4]\d)|(25[0-5]))$

Regex	Description
^	Beginning of line
(	A group consisting of
(	A subgroup consisting of
[0-1]?	Zero or one, both optional, followed by
\d	Any digit
{1,2}	Repeated one or two times, followed by
\.	A period
)	Close subgroup
|	Or
(	A subgroup consisting of
2	The digit *2*, followed by
[0-4]	Any digit from *0* to *4*, followed by
\d	Any digit, followed by
\.	A period
)	Close subgroup
|	Or
(	A subgroup consisting of
2	The digit *2*, followed by
5	The digit *5*, followed by

Table C-1. IP Address ^(([0-1]?\d{1,2}\.)|(2[0-4]\d\.)|(25[0-5]\.)){3} (([0-1]?\d{1,2})|(2[0-4]\d)|(25[0-5]))$ (Continued)

Regex	Description
[0-5]	Any digit from *0* to *5*, followed by
\.	A period
)	Close subgroup
)	Close group
{3}	Repeated three times, followed by
(	A group consisting of
(	A subgroup consisting of
[0-1]?	Zero or one, both optional, followed by
\d{1,2}	Any two digits
)	Close subgroup
|	Or
(	A subgroup consisting of
2	The digit *2*, followed by
[0-4]	Any digit from *0* to *4*, followed by
\d	Any digit
)	Close subgroup
|	Or
(	A subgroup consisting of
2	The digit *2*, followed by
5	The digit *5*, followed by
[0-5]	Any digit from *0* to *5*, followed by
)	Close subgroup
)	Subgroup
$	End of line

* **In English:** Three sets of three digits separated by periods, each ranging from 0 to 255, followed by three digits, each ranging from 0 to 255.

NOTE The following pattern also matches the IP address, but all the groups have been marked as noncapturing:

```
(?:(?:[0-1]?\d{1,2}\.)|(?:2[0-4]\d\.)|(?:25[0-5]\.)){3}(?:(?:[0-1]?\
d{1,2})|(?:2[0-4]\d)|(?:25[0-5]))
```

This is slightly more efficient than the previous pattern, but it's less legible.

Table C-2. Simple E-mail ***^(\p{Alnum}+(\.|_|\-)?)*\p{Alnum}@(\p{Alnum}+(\.|_|\-)?)*\p{Alpha}$***

Regex	Description
^	Beginning of line
(	A group consisting of
\p{Alnum}	A letter or a digit
+	Repeated one or more times, followed by
(	A subgroup consisting of
\.	A period
|	Or
_	An underscore
|	Or
\-	A hyphen
)	Close subgroup
?	The preceding punctuation is optional
)	Close group
*	The previous group can be repeated zero or more times, followed by
\p{Alnum}	A letter or a digit, followed by
@	An @ symbol, followed by
\p{Alnum}	A letter or a digit
+	Repeated one or more times, followed by

Table C-2. Simple E-mail ^(\p{Alnum}+(\.|_|\-)?)\p{Alnum}@(\p{Alnum}+(\.|_|\-)?)*\p{Alpha}$ (Continued)*

Regex	Description
(	A group consisting of
\.	A period
|	Or
_	An underscore
|	Or
\-	A hyphen
)	Close group
?	The preceding punctuation is optional
\p{Alpha}	An upper- or lowercase letter
$	End of line

* **In English:** Any number of alphanumeric characters followed by single hyphens, periods, or underscores, but ending in an alphanumeric character; followed by an @ symbol; followed by any number of alphanumeric characters; followed by single hyphens, periods, or underscores, but ending in an upper- or lowercase character.

NOTE The following pattern matches the previous one exactly, except that it allows an IP address as well:

```
^(\p{Alnum}+(\.|\_|\-)?)*\p{Alnum}@(((\p{Alnum}+(\.|\_|\-)?)*
\p{Alpha})|((([0-1]?\d{1,2}\.)|(2[0-4]\d\.)|(25[0-5]\.)){3}(([0-1]?\d{1,2})|
(2[0-4]\d)|(25[0-5]))))$
```

For a breakdown of the IP address pattern, please see Table C-1.

Table C-3. Digit Repeated Exactly n Times \d{n}, Where n Is the Number of Digits Needed

Regex	Description
\d	Any number
{n}	Repeated *n* times

* **In English:** *n* digits. Thus, if *n* was equal to 4, any four digits.

Table C-4. Characters Repeated Exactly n Times \w{n}, Where n Is the Number of Characters Needed

Regex	Description
\w	Any number, any digit, or an underscore symbol
{n}	Repeated *n* times

* **In English:** *n* characters. Thus, if *n* was equal to 4, any four characters.

Table C-5. Characters Repeated n to m Times \w{n.m}, Where n Is the Number of Characters Needed

Regex	Description
\w	Any number, any digit, or an underscore symbol
{n	Repeated *n* times
m}	But not more than *m* times

* **In English:** *n* characters. Thus, if *n* was equal to 4 and *m* was equal to 9, any four, five, six, seven, eight, or nine characters.

Table C-6. Credit Cards: Visa, MasterCard, American Express, and Discover
^((4\d{3})|(5[1-5]\d{2})|(6011))-?\d{4}-?\d{4}-?\d{4}|3[4,7]\d{13}$

Regex	Description
^	Beginning of line
(	A group consisting of
(	A subgroup consisting of
4	the digit four, followed by
\d[3]	Any three digits
)	Close subgroup
|	Or
(	A subgroup consisting of
5	The digit *5*, followed by
[1-5]	Any digit ranging from *1* to *5*, followed by
\d[2]	Any two digits
)	Close subgroup
|	Or
(	A subgroup consisting of
6001	The digits *6*, *0*, *0*, *1*
)	Close subgroup, followed by
-?	An optional hyphen, followed by
\d{4}	Any four digits, followed by
-?	An optional hyphen, followed by
\d{4}	Any four digits, followed by
-?	An optional hyphen, followed by
\d{4}	Any four digits, followed by
-?	An optional hyphen
|	Or
3	The digit *3*, followed by

Table C-6. Credit Cards: Visa, MasterCard, American Express, and Discover ^((4\d{3})|(5[1-5]\d{2})|(6011))-?\d{4}-?\d{4}-?\d{4}|3[4,7]\d{13}$ (Continued)

Regex	Description
[4,7]	A *4* or a *7*, followed by
\d{13]	Any thirteen digits, followed by
$	End of line

* **In English:** A number starting with *4* and three digits, or *5* and three digits, or *6011*, followed by a hyphen, followed by three sets of four digits, or 34 and thirteen digits, or 37 and thirteen digits.

NOTE This regex does not, and cannot, conform to mod 10 verification. To find a Java program that does, please visit http://www.influxs.com.

Table C-7. Real Number ^[+-]?\d+(\.\d+)?$

Regex	Description
^	Beginning of line, followed by
[+-]?	An optional plus or a minus sign
\d+	Followed by one or more digits, followed by
(	A group consisting of
\.	A period, followed by
\d+	One or more digits
)?	Close group, and make it optional
$	End of line

* **In English:** Any number of digits followed by an optional decimal component.

Index

L

M

Q

R

S

T

U

V

W

X

Z