Regex Metacharacters, Modes, and Constructs

The metacharacters and metasequences shown here represent most available types of regular expression constructs and their most common syntax. However, syntax and availability vary by implementation.

Character representations

Many implementations provide shortcuts to represent characters that may be difficult to input. (See MRE 115–118.)

Character shorthands
: Most implementations have specific shorthands for the `alert`, `backspace`, `escape character`, `form feed`, `newline`, `carriage return`, `horizontal tab`, and `vertical tab` characters. For example, `\n` is often a shorthand for the newline character, which is usually LF (012 octal), but can sometimes be CR (015 octal), depending on the operating system. Confusingly, many implementations use `\b` to mean both `backspace` and word boundary (position between a "word" character and a nonword character). For these implementations, `\b` means `backspace` in a character class (a set of possible characters to match in the string), and word boundary elsewhere.

Octal escape: `\num`
: Represents a character corresponding to a two- or three-digit octal number. For example, `\015\012` matches an ASCII CR/LF sequence.

Hex and Unicode escapes: `\x`*num*, `\x{`*num*`}`, `\u`*num*, `\U`*num*
: Represent characters corresponding to hexadecimal numbers. Four-digit and larger hex numbers can represent the range of Unicode characters. For example, `\x0D\x0A` matches an ASCII CR/LF sequence.

Control characters: `\c`*char*
: Corresponds to ASCII control characters encoded with values less than 32. To be safe, always use an uppercase *char*—some implementations do not handle lowercase representations. For example, `\cH` matches Control-H, an ASCII backspace character.

Character classes and class-like constructs

Character classes are used to specify a set of characters. A character class matches a single character in the input string that is within the defined set of characters. (See MRE 118–128.)

Normal classes: `[...]` *and* `[^...]`
: Character classes, `[...]`, and negated character classes, `[^...]`, allow you to list the characters that you do or do not want to match. A character class always matches one character. The - (dash) indicates a range of characters. For example, `[a-z]` matches any lowercase ASCII letter. To include the dash in the list of characters, either list it first, or escape it.

Almost any character: dot (`.`)
: Usually matches any character except a newline. However, the match mode usually can be changed so that dot also matches newlines. Inside a character class, dot matches just a dot.

Class shorthands: `\w`, `\d`, `\s`, `\W`, `\D`, `\S`
: Commonly provided shorthands for word character, digit, and space character classes. A word character is often all ASCII alphanumeric characters plus the underscore. However, the list of alphanumerics can include additional locale or Unicode alphanumerics, depending on the implementation. A lowercase shorthand (e.g., `\s`) matches a character from the class; uppercase (e.g., `\S`) matches a character not from the class. For example, `\d` matches a single digit character, and is usually equivalent to `[0-9]`.

POSIX character class: `[:`*alnum*`:]`
: POSIX defines several character classes that can be used only within regular expression character classes (see Table 1). Take, for example, `[:lower:]`. When written as `[[:lower:]]`, it is equivalent to `[a-z]` in the ASCII locale.

Table 1. POSIX character classes

Class	Meaning
`Alnum`	Letters and digits.
`Alpha`	Letters.
`Blank`	Space or tab only.
`Cntrl`	Control characters.
`Digit`	Decimal digits.
`Graph`	Printing characters, excluding space.
`Lower`	Lowercase letters.
`Print`	Printing characters, including space.
`Punct`	Printing characters, excluding letters and digits.
`Space`	Whitespace.
`Upper`	Uppercase letters.
`Xdigit`	Hexadecimal digits.

Unicode properties, scripts, and blocks: `\p{`*prop*`}`, `\P{`*prop*`}`

The Unicode standard defines classes of characters that have a particular property, belong to a script, or exist within a block. *Properties* are the character's defining characteristics, such as being a letter or a number (see Table 2). *Scripts* are systems of writing, such as Hebrew, Latin, or Han. *Blocks* are ranges of characters on the Unicode character map. Some implementations require that Unicode properties be prefixed with `Is` or `In`. For example, `\p{Ll}` matches lowercase letters in any Unicode-supported language, such as a or α.

Unicode combining character sequence: `\X`

Matches a Unicode base character followed by any number of Unicode-combining characters. This is a shorthand for `\P{M}\p{M}`. For example, `\X` matches è; as well as the two characters e'.

Table 2. Standard Unicode properties

Property	Meaning
`\p{L}`	Letters.
`\p{Ll}`	Lowercase letters.
`\p{Lm}`	Modifier letters.
`\p{Lo}`	Letters, other. These have no case, and are not considered modifiers.
`\p{Lt}`	Titlecase letters.
`\p{Lu}`	Uppercase letters.
`\p{C}`	Control codes and characters not in other categories.
`\p{Cc}`	ASCII and Latin-1 control characters.
`\p{Cf}`	Nonvisible formatting characters.
`\p{Cn}`	Unassigned code points.
`\p{Co}`	Private use, such as company logos.
`\p{Cs}`	Surrogates.
`\p{M}`	Marks meant to combine with base characters, such as accent marks.
`\p{Mc}`	Modification characters that take up their own space. Examples include "vowel signs."
`\p{Me}`	Marks that enclose other characters, such as circles, squares, and diamonds.
`\p{Mn}`	Characters that modify other characters, such as accents and umlauts.
`\p{N}`	Numeric characters.
`\p{Nd}`	Decimal digits in various scripts.
`\p{Nl}`	Letters that represent numbers, such as Roman numerals.
`\p{No}`	Superscripts, symbols, or nondigit characters representing numbers.
`\p{P}`	Punctuation.
`\p{Pc}`	Connecting punctuation, such as an underscore.
`\p{Pd}`	Dashes and hyphens.
`\p{Pe}`	Closing punctuation complementing `\p{Ps}`.
`\p{Pi}`	Initial punctuation, such as opening quotes.

Table 2. Standard Unicode properties (continued)

Property	Meaning
`\p{Pf}`	Final punctuation, such as closing quotes.
`\p{Po}`	Other punctuation marks.
`\p{Ps}`	Opening punctuation, such as opening parentheses.
`\p{S}`	Symbols.
`\p{Sc}`	Currency.
`\p{Sk}`	Combining characters represented as individual characters.
`\p{Sm}`	Math symbols.
`\p{So}`	Other symbols.
`\p{Z}`	Separating characters with no visual representation.
`\p{Zl}`	Line separators.
`\p{Zp}`	Paragraph separators.
`\p{Zs}`	Space characters.

Anchors and zero-width assertions

Anchors and "zero-width assertions" match positions in the input string. (See MRE 128–134.)

Start of line/string: `^`, `\A`
: Matches at the beginning of the text being searched. In multiline mode, `^` matches after any newline. Some implementations support `\A`, which matches only at the beginning of the text.

End of line/string: `$`, `\Z`, `\z`
: `$` matches at the end of a string. In multiline mode, `$` matches before any newline. When supported, `\Z` matches the end of string or the point before a string-ending newline, regardless of match mode. Some implementations also provide `\z`, which matches only the end of the string, regardless of newlines.

Start of match: `\G`
: In iterative matching, `\G` matches the position where the previous match ended. Often, this spot is reset to the beginning of a string on a failed match.

Word boundary: `\b`, `\B`, `\<`, `\>`
: Word boundary metacharacters match a location where a word character is next to a nonword character. `\b` often specifies a word boundary location, and `\B` often specifies a not-word-boundary location. Some implementations provide separate metasequences for start- and end-of-word boundaries, often `\<` and `\>`.

Lookahead: `(?=...)`, `(?!...)`
Lookbehind: `(?<=...)`, `(?<!...)`
: *Lookaround constructs* match a location in the text where the subpattern would match (lookahead), would not match (negative lookahead), would have finished matching (lookbehind), or would not have finished matching (negative lookbehind). For example, `foo(?=bar)` matches `foo` in `foobar`, but not `food`. Implementations often limit lookbehind constructs to subpatterns with a predetermined length.

Comments and mode modifiers

Mode modifiers change how the regular expression engine interprets a regular expression. (See MRE 110–113, 135–136.)

Multiline mode: `m`
: Changes the behavior of `^` and `$` to match next to newlines within the input string.

Single-line mode: `s`
: Changes the behavior of `.` (dot) to match all characters, including newlines, within the input string.

Case-insensitive mode: `i`
: Treat letters that differ only in case as identical.

Free-spacing mode: `x`
: Allows for whitespace and comments within a regular expression. The whitespace and comments (starting with `#` and extending to the end of the line) are ignored by the regular expression engine.

Mode modifiers: `(?i)`, `(?-i)`, `(?mod:...)`
: Usually, mode modifiers may be set within a regular expression with `(?mod)` to turn modes on for the rest of the current subexpression; `(?-mod)` to turn modes off for the rest of the current subexpression; and `(?mod:...)` to turn modes on or off between the colon and the closing parentheses. For example, `use (?i:perl)` matches `use perl`, `use Perl`, `use PeRl`, etc.

Comments: `(?#...)` *and* `#`
: In free-spacing mode, `#` indicates that the rest of the line is a comment. When supported, the comment span `(?#...)` can be embedded anywhere in a regular expression, regardless of mode. For example, `.{0,80}(?#Field limit is 80 chars)` allows you to make notes about why you wrote `.{0,80}`.

Literal-text span: `\Q...\E`
: Escapes metacharacters between `\Q` and `\E`. For example, `\Q(.*)\E` is the same as `\(\.\*\)`.

Grouping, capturing, conditionals, and control

This section covers syntax for grouping subpatterns, capturing submatches, conditional submatches, and quantifying the number of times a subpattern matches. (See MRE 137–142.)

Capturing and grouping parentheses: `(...)` *and* `\1`, `\2`, *etc.*
: Parentheses perform two functions: grouping and capturing. Text matched by the subpattern within parentheses is captured for later use. Capturing parentheses are numbered by counting their opening parentheses from the left. If backreferences are available, the submatch can be referred to later in the same match with `\1`, `\2`, etc. The captured text is made available after a match by implementation-specific methods. For example, `\b(\w+)\b\s+\1\b` matches duplicate words, such as `the the`.

Grouping-only parentheses: `(?:...)`
: Groups a subexpression, possibly for alternation or quantifiers, but does not capture the submatch. This is useful for efficiency and reusability. For example, `(?:foobar)` matches `foobar`, but does not save the match to a capture group.

Named capture: `(?<name>...)`
: Performs capturing and grouping, with captured text later referenced by *name*. For example, `Subject:(?<subject>.*)` captures the text following `Subject:` to a capture group that can be referenced by the name `subject`.

Atomic grouping: `(?>...)`
: Text matched within the group is never backtracked into, even if this leads to a match failure. For example, `(?>[ab]*)\w\w` matches `aabbcc`, but not `aabbaa`.

Alternation: `...|...`
: Allows several subexpressions to be tested. Alternation's low precedence sometimes causes subexpressions to be longer than intended, so use parentheses to specifically group what you want alternated. Thus, `\b(foo|bar)\b` matches the words `foo` or `bar`.

Conditional: `(?(if)then|else)`
: The *if* is implementation-dependent, but generally is a reference to a captured subexpression or a lookaround. The *then* and *else* parts are both regular expression patterns. If the *if* part is true, the *then* is applied. Otherwise, *else* is applied. For example, `(<)?foo(?(1)>|bar)` matches `<foo>` as well as `foobar`.

Greedy quantifiers: `*`, `+`, `?`, `{num,num }`
: The greedy quantifiers determine how many times a construct may be applied. They attempt to match as many times as possible, but will backtrack and give up matches if necessary for the success of the overall match. For example, `(ab)+` matches all of `abababababab`.

Lazy quantifiers: `*?`, `+?`, `??`, `{num,num }?`
: Lazy quantifiers control how many times a construct may be applied. However, unlike greedy quantifiers, they attempt to match as few times as possible. For example, `(an)+?` matches only `an` of `banana`.

Possessive quantifiers: `*+`, `++`, `?+`, `{num,num }+`
: Possessive quantifiers are like greedy quantifiers, except that they "lock in" their match, disallowing later backtracking to break up the submatch. For example, `(ab)++ab` will not match `abababab`.

Unicode Support

The *Unicode* character set gives unique numbers to the characters in all the world's languages. Because of the large number of possible characters, Unicode requires more than one byte to represent a character. Some regular expression implementations will not understand Unicode characters because they expect 1 byte ASCII characters. Basic support for Unicode characters starts with the ability to match a literal string of Unicode characters. Advanced support includes character classes and other constructs that incorporate characters from all Unicode-supported languages. For example, `\w` might match `è;` as well as `e`.

Regular Expression Cookbook

This section contains simple versions of common regular expression patterns. You may need to adjust them to meet your needs.

Each expression is presented here with target strings that it matches, and target strings that it does not match, so you can get a sense of what adjustments you may need to make for your own use cases.

They are written in the Perl style:

```
/pattern/mode
s/pattern/replacement/mode
```


Recipes

Removing leading and trailing whitespace

```
s/^\s+//
s/\s+$//
```

Matches: `" foo bar "`, `"foo "`

Nonmatches: `"foo bar"`

Numbers from 0 to 999999

```
/^\d{1,6}$/
```

Matches: `42`, `678234`

Nonmatches: `10,000`

Valid HTML Hex code

```
/^#([a-fA-F0-9]){3}(([a-fA-F0-9]){3})?$/
```

Matches: `#fff`, `#1a1`, `#996633`

Nonmatches: `#ff`, `FFFFFF`

U.S. Social Security number

```
/^\d{3}-\d{2}-\d{4}$/
```

Matches: `078-05-1120`

Nonmatches: `078051120`, `1234-12-12`

U.S. zip code

```
/^\d{5}(-\d{4})?$/
```

Matches: `94941-3232`, `10024`

Nonmatches: `949413232`

U.S. currency

```
/^\$\(d{1,3}(\,\d{3})*|\d+)(\.\d{2})?$/
```

Matches: `$20`, `$15,000.01`

Nonmatches: `$1.001`, `$.99`

Match date: MM/DD/YYYY HH:MM:SS

```
/^\d\d\/\d\d\/\d\d\d\d \d\d:\d\d:\d\d$/
```

Matches: `04/30/1978 20:45:38`

Nonmatches: `4/30/1978 20:45:38`, `4/30/78`

Leading pathname

```
/^.*\//
```

Matches: `/usr/local/bin/apachectl`

Nonmatches: `C:\\System\foo.exe`

(See MRE 190–192.)

Dotted Quad IP address

```
/^(\d|[01]?\d\d|2[0-4]\d|25[0-5])\.(\d|[01]?\d\d|2[0-4]
\d|25[0-5])\.
(\d|[01]?\d\d|2[0-4]\d|25[0-5])\.(\d|[01]?\d\d|2[0-4]
\d|25[0-5])$/
```

Matches: `127.0.0.1`, `224.22.5.110`

Nonmatches: `127.1`

(See MRE 187–189.)

MAC address

```
/^([0-9a-fA-F]{2}:){5}[0-9a-fA-F]{2}$/
```

Matches: `01:23:45:67:89:ab`

Nonmatches: `01:23:45`, `0123456789ab`

Email

```
/^[0-9a-zA-Z]([-.\w]*[0-9a-zA-Z_+])*@([0-9a-zA-Z][-\w]*
[0-9a-zA-Z]\.)+[a-zA-Z]{2,9}$/
```

Matches: *tony@example.com*, *tony@i-e.com*, `tony@mail.example.museum`

Nonmatches: `.@example.com`, *tony@i-.com*, *tony@example.a*

(See MRE 70.)

Java (java.util.regex)

Java 1.4 introduced regular expressions with Sun's `java.util.regex` package. Although there are competing packages available for previous versions of Java, Sun's is now the standard. Sun's package uses a Traditional NFA match engine. For an explanation of the rules behind a Traditional NFA engine, see "Introduction to Regexes and Pattern Matching." This section covers regular expressions in Java 1.5 and 1.6.

Supported Metacharacters

`java.util.regex` supports the metacharacters and metasequences listed in Table 11 through Table 15. For expanded definitions of each metacharacter, see "Regex Metacharacters, Modes, and Constructs."

Table 11. Java character representations

Sequence	Meaning
`\a`	Alert (bell).
`\b`	Backspace, `\x08`, supported only in character class.
`\e`	Esc character, `\x1B`.
`\n`	Newline, `\x0A`.
`\r`	Carriage return, `\x0D`.
`\f`	Form feed, `\x0C`.
`\t`	Horizontal tab, `\x09`.
`\0`*octal*	Character specified by a one-, two-, or three-digit octal code.
`\x`*hex*	Character specified by a two-digit hexadecimal code.
`\u`*hex*	Unicode character specified by a four-digit hexadecimal code.
`\c`*char*	Named control character.

Table 12. Java character classes and class-like constructs

Class	Meaning
`[...]`	A single character listed or contained in a listed range.
`[^...]`	A single character not liste and not contained within a listed range.
`.`	Any character, except a line terminator (unless `DOTALL` mode).
`\w`	Word character, `[a-zA-Z0-9_]`.
`\W`	Nonword character, `[^a-zA-Z0-9_]`.
`\d`	Digit, `[0-9]`.
`\D`	Nondigit, `[^0-9]`.
`\s`	Whitespace character, `[ \t\n\f\r\x0B]`.
`\S`	Nonwhitespace character, `[^ \t\n\f\r\x0B]`.
`\p{`*prop*`}`	Character contained by given POSIX character class, Unicode property, or Unicode block.
`\P{`*prop*`}`	Character not contained by given POSIX character class, Unicode property, or Unicode block.

Table 13. Java anchors and other zero-width tests

Sequence	Meaning
`^`	Start of string, or the point after any newline if in `MULTILINE` mode.
`\A`	Beginning of string, in any match mode.
`$`	End of string, or the point before any newline if in `MULTILINE` mode.
`\Z`	End of string, but before any final line terminator, in any match mode.
`\z`	End of string, in any match mode.
`\b`	Word boundary.
`\B`	Not-word-boundary.
`\G`	Beginning of current search.

Table 13. Java anchors and other zero-width tests (continued)

Sequence	Meaning
`(?=...)`	Positive lookahead.
`(?!...)`	Negative lookahead.
`(?<=...)`	Positive lookbehind.
`(?<!...)`	Negative lookbehind.

Table 14. Java comments and mode modifiers

Modifier/sequence	Mode character	Meaning
`Pattern.UNIX_LINES`	`d`	Treat `\n` as the only line terminator.
`Pattern.DOTALL`	`s`	Dot (`.`) matches any character, including a line terminator.
`Pattern.MULTILINE`	`m`	`^` and `$` match next to embedded line terminators.
`Pattern.COMMENTS`	`x`	Ignore whitespace, and allow embedded comments starting with `#`.
`Pattern.CASE_INSENSITIVE`	`i`	Case-insensitive match for ASCII characters.
`Pattern.UNICODE_CASE`	`u`	Case-insensitive match for Unicode characters.
`Pattern.CANON_EQ`		Unicode "canonical equivalence" mode, where characters, or sequences of a base character and combining characters with identical visual representations, are treated as equals.
`(?`*mode*`)`		Turn listed modes (one or more of `idmsux`) on for the rest of the subexpression.
`(?-`*mode*`)`		Turn listed modes (one or more of `idmsux`) off for the rest of the subexpression.
`(?`*mode*`:...)`		Turn listed modes (one or more of `idmsux`) on within parentheses.

Table 14. Java comments and mode modifiers (continued)

Modifier/sequence	Mode character	Meaning
`(?-mode:...)`		Turn listed modes (one or more of `idmsux`) off within parentheses.
`#...`		Treat rest of line as a comment in `/x` mode.

Table 15. Java grouping, capturing, conditional, and control

Sequence	Meaning
`(...)`	Group subpattern and capture submatch into `\1`,`\2`,… and `$1`, `$2`,….
`\n`	Contains text matched by the *n*th capture group.
`$n`	In a replacement string, contains text matched by the *n*th capture group.
`(?:...)`	Groups subpattern, but does not capture submatch.
`(?>...)`	Atomic grouping.
`...|...`	Try subpatterns in alternation.
`*`	Match 0 or more times.
`+`	Match 1 or more times.
`?`	Match 1 or 0 times.
`{n}`	Match exactly *n* times.
`{n,}`	Match at least *n* times.
`{x,y}`	Match at least *x* times, but no more than *y* times.
`*?`	Match 0 or more times, but as few times as possible.
`+?`	Match 1 or more times, but as few times as possible.
`??`	Match 0 or 1 times, but as few times as possible.
`{n,}?`	Match at least *n* times, but as few times as possible.
`{x,y}?`	Match at least *x* times, no more than *y* times, and as few times as possible.
`*+`	Match 0 or more times, and never backtrack.
`++`	Match 1 or more times, and never backtrack.

Table 15. Java grouping, capturing, conditional, and control (continued)

Sequence	Meaning
`?+`	Match 0 or 1 times, and never backtrack.
`{n}+`	Match at least *n* times, and never backtrack.
`{n,}+`	Match at least *n* times, and never backtrack.
`{x,y}+`	Match at least *x* times, no more than *y* times, and never backtrack.

Regular Expression Classes and Interfaces

Regular expression functions are contained in two main classes, `java.util.regex.Pattern` and `java.util.regex.Matcher`; an exception, `java.util.regex.PatternSyntaxException`; and an interface, `CharSequence`. Additionally, the `String` class implements the `CharSequence` interface to provide basic pattern-matching methods. `Pattern` objects are compiled regular expressions that can be applied to any `CharSequence`. A `Matcher` is a stateful object that scans for one or more occurrences of a `Pattern` applied in a string (or any object implementing `CharSequence`).

Backslashes in regular expression `String` literals need to be escaped. So, `\n` (newline) becomes `\\n` when used in a Java `String` literal that is to be used as a regular expression.

java.lang.String

Description

Methods for pattern matching.

Methods

`boolean matches(String regex)`
: Return true if *regex* matches the entire `String`.

`String[ ] split(String regex)`
: Return an array of the substrings surrounding matches of *regex*.

`String [ ] split(String regex, int limit)`
: Return an array of the substrings surrounding the first *limit*-1 matches of *regex*.

`String replaceFirst(String regex, String replacement)`
: Replace the substring matched by *regex* with *replacement*.

`String replaceAll(String regex, String replacement)`
: Replace all substrings matched by *regex* with *replacement*.

java.util.regex.Pattern

Description

Models a regular expression pattern.

Methods

`static Pattern compile(String regex)`
: Construct a `Pattern` object from *regex*.

`static Pattern compile(String regex, int flags)`
: Construct a new `Pattern` object out of *regex*, and the OR'd mode-modifier constants *flags*.

`int flags( )`
: Return the `Pattern`'s mode modifiers.

`Matcher matcher(CharSequence input)`
: Construct a `Matcher` object that will match this `Pattern` against *input*.

`static boolean matches(String regex, CharSequence input)`
: Return true if *regex* matches the entire string *input*.

`String pattern( )`
: Return the regular expression used to create this `Pattern`.

`static String quote(String text)`
: Escapes the text so that regular expression operators will be matched literally.

`String[ ] split(CharSequence input)`
: Return an array of the substrings surrounding matches of this `Pattern` in *input*.

`String[ ] split(CharSequence input, int limit)`
: Return an array of the substrings surrounding the first *limit* matches of this pattern in *regex*.

java.util.regex.Matcher

Description

Models a stateful regular expression pattern matcher and pattern matching results.

Methods

`Matcher appendReplacement(StringBuffer sb, String replacement)`
: Append substring preceding match and *replacement* to *sb*.

`StringBuffer appendTail(StringBuffer sb)`
: Append substring following end of match to *sb*.

`int end( )`
: Index of the first character after the end of the match.

`int end(int group)`
: Index of the first character after the text captured by *group*.

`boolean find( )`
: Find the next match in the input string.

`boolean find(int start)`
: Find the next match after character position *start*.

`String group( )`
: Text matched by this `Pattern`.

`String group(int group)`
: Text captured by capture group *group*.

`int groupCount( )`
: Number of capturing groups in `Pattern`.

`boolean hasAnchoringBounds( )`
: Return true if this `Matcher` uses anchoring bounds so that anchor operators match at the region boundaries, not just at the start and end of the target string.

`boolean hasTransparentBounds( )`
: True if this `Matcher` uses transparent bounds so that lookaround operators can see outside the current search bounds. Defaults to false.

`boolean hitEnd( )`
: True if the last match attempts to inspect beyond the end of the input. In scanners, this is an indication that more input may have resulted in a longer match.

`boolean lookingAt( )`
: True if the pattern matches at the beginning of the input.

`boolean matches( )`
: Return true if `Pattern` matches entire input string.

`Pattern pattern( )`
: Return `Pattern` object used by this `Matcher`.

`static String quoteReplacement(String string)`
: Escape special characters evaluated during replacements.

`Matcher region(int start, int end)`
: Return this matcher and run future matches in the region between *`start`* characters and *`end`* characters from the beginning of the string.

`int regionStart( )`
: Return the starting offset of the search region. Defaults to zero.

`int regionEnd( )`
: Return the ending offset of the search region. Defaults to the length of the target string.

`String replaceAll(String replacement)`
: Replace every match with *`replacement`*.

`String replaceFirst(String replacement)`
: Replace first match with *`replacement`*.

`boolean requireEnd( )`
: Return true if the success of the last match relied on the end of the input. In scanners, this is an indication that more input may have caused a failed match.

`Matcher reset( )`
: Reset this matcher so that the next match starts at the beginning of the input string.

`Matcher reset(CharSequence input)`
: Reset this matcher with new *`input`*.

`int start( )`
: Index of first character matched.

`int start(int group)`
: Index of first character matched in captured substring *`group`*.

`MatchResult toMatchResult( )`
: Return a `MatchResult` object for the most recent match.

`String toString( )`
: Return a string representation of the matcher for debugging.

`Matcher useAnchorBounds(boolean b)`
: If true, set the `Matcher` to use anchor bounds so that anchor operators match at the beginning and end of the current search bounds, rather than the beginning and end of the search string. Defaults to true.

`Matcher usePattern(Pattern p)`
: Replace the `Matcher`'s pattern, but keep the rest of the match state.

`Matcher useTransparentBounds(boolean b)`
: If true, set the `Matcher` to use transparent bounds so that lookaround operators can see outside of the current search bounds. Defaults to false.

java.util.regex.PatternSyntaxException

Description

Thrown to indicate a syntax error in a regular expression pattern.

Methods

`PatternSyntaxException(String desc, String regex, int index)`
: Construct an instance of this class.

`String getDescription( )`
: Return error description.

`int getIndex( )`
: Return error index.

`String getMessage( )`
: Return a multiline error message containing error description, index, regular expression pattern, and indication of the position of the error within the pattern.

`String getPattern( )`
: Return the regular expression pattern that threw the exception.

java.lang.CharSequence

Description

Defines an interface for read-only access so that regular expression patterns may be applied to a sequence of characters.

Methods

`char charAt(int index)`
: Return the character at the zero-based position *index*.

`int length( )`
: Return the number of characters in the sequence.

`CharSequence subSequence(int start, int end)`
: Return a subsequence, including the *start* index, and excluding the *end* index.

`String toString( )`
: Return a `String` representation of the sequence.

Unicode Support

This package supports Unicode 4.0, although `\w`, `\W`, `\d`, `\D`, `\s`, and `\S` support only ASCII. You can use the equivalent Unicode properties `\p{L}`, `\P{L}`, `\p{Nd}`, `\P{Nd}`, `\p{Z}`, and `\P{Z}`. The word boundary sequences—`\b` and `\B`—do understand Unicode.

For supported Unicode properties and blocks, see Table 2. This package supports only the short property names, such as `\p{Lu}`, and not `\p{Lowercase_Letter}`. Block names require the `In` prefix, and support only the name form without spaces or underscores, for example, `\p{InGreekExtended}`, not `\p{In_Greek_Extended}` or `\p{In Greek Extended}`.

Examples

Example 5. Simple match

```
import java.util.regex.*;

// Find Spider-Man, Spiderman, SPIDER-MAN, etc.
public class StringRegexTest {
  public static void main(String[] args) throws Exception {
    String dailyBugle = "Spider-Man Menaces City!";

    //regex must match entire string
```


Example 5. Simple match (continued)

```
    String regex = "(?i).*spider[- ]?man.*";

    if (dailyBugle.matches(regex)) {
      System.out.println("Matched: " + dailyBugle);
    }
  }
}
```

Example 6. Match and capture group

```
// Match dates formatted like MM/DD/YYYY, MM-DD-YY,...
import java.util.regex.*;

public class MatchTest {
  public static void main(String[] args) throws Exception {
    String date = "12/30/1969";
    Pattern p =
      Pattern.compile("^(\\d\\d)[-/](\\d\\d)[-/](\\d\\d(?:\\d\
\d)?)$");

    Matcher m = p.matcher(date);

    if (m.find( )) {
      String month = m.group(1);
      String day   = m.group(2);
      String year  = m.group(3);
      System.out.printf("Found %s-%s-%s\n", year, month, day);
    }
  }
}
```

Example 7. Simple substitution

```
// Example -. Simple substitution
// Convert <br> to <br /> for XHTML compliance
import java.util.regex.*;

public class SimpleSubstitutionTest {
  public static void main(String[] args) {
    String text = "Hello world. <br>";
```

Example 7. Simple substitution (continued)

```
    Pattern p = Pattern.compile("<br>", Pattern.CASE_
INSENSITIVE);
    Matcher m = p.matcher(text);

    String result = m.replaceAll("<br />");
    System.out.println(result);
  }
}
```

Example 8. Harder substitution

```
// urlify - turn URLs into HTML links
import java.util.regex.*;

public class Urlify {
  public static void main (String[  ] args) throws Exception {
   String text = "Check the web site, http://www.oreilly.com/
catalog/regexppr.";
   String regex =
        "\\b                        # start at word\n"
     +  "                           # boundary\n"
     +  "(                          # capture to $1\n"
     +  "(https?|telnet|gopher|file|wais|ftp) : \n"
     +  "                           # resource and colon\n"
     +  "[\\w/\\#~:.?+=&%@!\\-] +?   # one or more valid\n"
     +  "                           # characters\n"
     +  "                           # but take as little\n"
     +  "                           # as possible\n"
     +  ")\n"
     +  "(?=                        # lookahead\n"
     +  "[.:?\\-] *                 # for possible punc\n"
     +  "(?: [^\\w/\\#~:.?+=&%@!\\-] # invalid character\n"
     +  "| $ )                      # or end of string\n"
     +  ")";

    Pattern p = Pattern.compile(regex,
        Pattern.CASE_INSENSITIVE + Pattern.COMMENTS);
    Matcher m = p.matcher(text);
    String result = m.replaceAll("<a href=\"$1\">$1</a>");
    System.out.println(result);
  }
}
```