Java SE 17 & JDK 17

Module java.base
Package java.util.regex

Class Pattern

java.lang.Object
    java.util.regex.Pattern

All Implemented Interfaces:

`Serializable`

```
public final class Pattern
extends Object
implements Serializable
```

A compiled representation of a regular expression.

A regular expression, specified as a string, must first be compiled into an instance of this class. The resulting pattern can then be used to create a `Matcher` object that can match arbitrary character sequences against the regular expression. All of the state involved in performing a match resides in the matcher, so many matchers can share the same pattern.

A typical invocation sequence is thus

```
    Pattern p = Pattern.compile("a*b");
    Matcher m = p.matcher("aaaaab");
    boolean b = m.matches();
```

A `matches` method is defined by this class as a convenience for when a regular expression is used just once. This method compiles an expression and matches an input sequence against it in a single invocation. The statement

```
    boolean b = Pattern.matches("a*b", "aaaaab");
```

is equivalent to the three statements above, though for repeated matches it is less efficient since it does not allow the compiled pattern to be reused.

Instances of this class are immutable and are safe for use by multiple concurrent threads. Instances of the `Matcher` class are not safe for such use.

Summary of regular-expression constructs

Construct	Matches
Characters	
x	The character *x*
`\\`	The backslash character
`\0`*n*	The character with octal value `0`*n* (0 <= *n* <= 7)

Construct	**Matches**
`\0`*nn*	The character with octal value `0`*nn* (0 <= *n* <= 7)
`\0`*mnn*	The character with octal value `0`*mnn* (0 <= *m* <= 3, 0 <= *n* <= 7)
`\x`*hh*	The character with hexadecimal value `0x`*hh*
`\u`*hhhh*	The character with hexadecimal value `0x`*hhhh*
`\x`*{h...h}*	The character with hexadecimal value `0x`*h...h* (`Character.MIN_CODE_POINT` <= `0x`*h...h* <= `Character.MAX_CODE_POINT`)
`\N{`*name*`}`	The character with Unicode character name '*name*'
`\t`	The tab character (`'\u0009'`)
`\n`	The newline (line feed) character (`'\u000A'`)
`\r`	The carriage-return character (`'\u000D'`)
`\f`	The form-feed character (`'\u000C'`)
`\a`	The alert (bell) character (`'\u0007'`)
`\e`	The escape character (`'\u001B'`)
`\c`*x*	The control character corresponding to *x*
Character classes	
`[abc]`	`a`, `b`, or `c` (simple class)
`[^abc]`	Any character except `a`, `b`, or `c` (negation)
`[a-zA-Z]`	`a` through `z` or `A` through `Z`, inclusive (range)
`[a-d[m-p]]`	`a` through `d`, or `m` through `p`: `[a-dm-p]` (union)
`[a-z&&[def]]`	`d`, `e`, or `f` (intersection)
`[a-z&&[^bc]]`	`a` through `z`, except for `b` and `c`: `[ad-z]` (subtraction)
`[a-z&&[^m-p]]`	`a` through `z`, and not `m` through `p`: `[a-lq-z]`(subtraction)
Predefined character classes	
`.`	Any character (may or may not match line terminators)
`\d`	A digit: `[0-9]`
`\D`	A non-digit: `[^0-9]`
`\h`	A horizontal whitespace character: `[ \t\xA0\u1680\u180e\u2000-\u200a\u202f\u205f\u3000]`
`\H`	A non-horizontal whitespace character: `[^\h]`
`\s`	A whitespace character: `[ \t\n\x0B\f\r]`
`\S`	A non-whitespace character: `[^\s]`
`\v`	A vertical whitespace character: `[\n\x0B\f\r\x85\u2028\u2029]`
`\V`	A non-vertical whitespace character: `[^\v]`
`\w`	A word character: `[a-zA-Z_0-9]`

Construct	Matches
\W	A non-word character: [^\w]
POSIX character classes (US-ASCII only)	
\p{Lower}	A lower-case alphabetic character: [a-z]
\p{Upper}	An upper-case alphabetic character:[A-Z]
\p{ASCII}	All ASCII:[\x00-\x7F]
\p{Alpha}	An alphabetic character:[\p{Lower}\p{Upper}]
\p{Digit}	A decimal digit: [0-9]
\p{Alnum}	An alphanumeric character:[\p{Alpha}\p{Digit}]
\p{Punct}	Punctuation: One of !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
\p{Graph}	A visible character: [\p{Alnum}\p{Punct}]
\p{Print}	A printable character: [\p{Graph}\x20]
\p{Blank}	A space or a tab: [\t]
\p{Cntrl}	A control character: [\x00-\x1F\x7F]
\p{XDigit}	A hexadecimal digit: [0-9a-fA-F]
\p{Space}	A whitespace character: [\t\n\x0B\f\r]
java.lang.Character classes (simple java character type)	
\p{javaLowerCase}	Equivalent to java.lang.Character.isLowerCase()
\p{javaUpperCase}	Equivalent to java.lang.Character.isUpperCase()
\p{javaWhitespace}	Equivalent to java.lang.Character.isWhitespace()
\p{javaMirrored}	Equivalent to java.lang.Character.isMirrored()
Classes for Unicode scripts, blocks, categories and binary properties	
\p{IsLatin}	A Latin script character (script)
\p{InGreek}	A character in the Greek block (block)
\p{Lu}	An uppercase letter (category)
\p{IsAlphabetic}	An alphabetic character (binary property)
\p{Sc}	A currency symbol
\P{InGreek}	Any character except one in the Greek block (negation)
[\p{L}&&[^\p{Lu}]]	Any letter except an uppercase letter (subtraction)
Boundary matchers	
^	The beginning of a line
$	The end of a line
\b	A word boundary
\b{g}	A Unicode extended grapheme cluster boundary

Construct	Matches
`\B`	A non-word boundary
`\A`	The beginning of the input
`\G`	The end of the previous match
`\Z`	The end of the input but for the final terminator, if any
`\z`	The end of the input
Linebreak matcher	
`\R`	Any Unicode linebreak sequence, is equivalent to `\u000D\u000A|[\u000A\u000B\u000C\u000D\u0085\u2028\u2029]`
Unicode Extended Grapheme matcher	
`\X`	Any Unicode extended grapheme cluster
Greedy quantifiers	
X`?`	*X*, once or not at all
X`*`	*X*, zero or more times
X`+`	*X*, one or more times
X`{`*n*`}`	*X*, exactly *n* times
X`{`*n*`,}`	*X*, at least *n* times
X`{`*n*`,`*m*`}`	*X*, at least *n* but not more than *m* times
Reluctant quantifiers	
X`??`	*X*, once or not at all
X`*?`	*X*, zero or more times
X`+?`	*X*, one or more times
X`{`*n*`}?`	*X*, exactly *n* times
X`{`*n*`,}?`	*X*, at least *n* times
X`{`*n*`,`*m*`}?`	*X*, at least *n* but not more than *m* times
Possessive quantifiers	
X`?+`	*X*, once or not at all
X`*+`	*X*, zero or more times
X`++`	*X*, one or more times
X`{`*n*`}+`	*X*, exactly *n* times
X`{`*n*`,}+`	*X*, at least *n* times
X`{`*n*`,`*m*`}+`	*X*, at least *n* but not more than *m* times
Logical operators	

Construct	Matches
XY	*X* followed by *Y*
X`|`*Y*	Either *X* or *Y*
`(`*X*`)`	X, as a capturing group
Back references	
`\`*n*	Whatever the *n*th capturing group matched
`\`*k*`<`*name*`>`	Whatever the named-capturing group "name" matched
Quotation	
`\`	Nothing, but quotes the following character
`\Q`	Nothing, but quotes all characters until `\E`
`\E`	Nothing, but ends quoting started by `\Q`
Special constructs (named-capturing and non-capturing)	
`(?<name>`*X*`)`	*X*, as a named-capturing group
`(?:`*X*`)`	*X*, as a non-capturing group
`(?idmsuxU-idmsuxU)`	Nothing, but turns match flags i d m s u x U on - off
`(?idmsuxU-idmsuxU:`*X*`)`	*X*, as a non-capturing group with the given flags i d m s u x U on - off
`(?=`*X*`)`	*X*, via zero-width positive lookahead
`(?!`*X*`)`	*X*, via zero-width negative lookahead
`(?<=`*X*`)`	*X*, via zero-width positive lookbehind
`(?<!`*X*`)`	*X*, via zero-width negative lookbehind
`(?>`*X*`)`	*X*, as an independent, non-capturing group

Backslashes, escapes, and quoting

The backslash character (`'\'`) serves to introduce escaped constructs, as defined in the table above, as well as to quote characters that otherwise would be interpreted as unescaped constructs. Thus the expression `\\` matches a single backslash and `\{` matches a left brace.

It is an error to use a backslash prior to any alphabetic character that does not denote an escaped construct; these are reserved for future extensions to the regular-expression language. A backslash may be used prior to a non-alphabetic character regardless of whether that character is part of an unescaped construct.

Backslashes within string literals in Java source code are interpreted as required by *The Java Language Specification* as either Unicode escapes (section 3.3) or other character escapes (section 3.10.6) It is therefore necessary to double backslashes in string literals that represent regular expressions to protect them from interpretation by the Java bytecode compiler. The string literal `"\b"`, for example, matches a single backspace character when interpreted as a regular expression, while `"\\b"` matches a word boundary. The string literal `"\(hello\)"` is illegal and leads to a compile-time error; in order to match the string `(hello)` the string literal `"\\(hello\\)"` must be used.

Character Classes

Character classes may appear within other character classes, and may be composed by the union operator (implicit) and the intersection operator (&&). The union operator denotes a class that contains every character that is in at least one of its operand classes. The intersection operator denotes a class that contains every character that is in both of its operand classes.

The precedence of character-class operators is as follows, from highest to lowest:

Precedence	Name	Example
1	Literal escape	`\x`
2	Grouping	`[...]`
3	Range	`a-z`
4	Union	`[a-e][i-u]`
5	Intersection	`[a-z&&[aeiou]]`

Note that a different set of metacharacters are in effect inside a character class than outside a character class. For instance, the regular expression `.` loses its special meaning inside a character class, while the expression `-` becomes a range forming metacharacter.

Line terminators

A *line terminator* is a one- or two-character sequence that marks the end of a line of the input character sequence. The following are recognized as line terminators:

- A newline (line feed) character (`'\n'`),
- A carriage-return character followed immediately by a newline character (`"\r\n"`),
- A standalone carriage-return character (`'\r'`),
- A next-line character (`'\u0085'`),
- A line-separator character (`'\u2028'`), or
- A paragraph-separator character (`'\u2029'`).

If `UNIX_LINES` mode is activated, then the only line terminators recognized are newline characters.

The regular expression `.` matches any character except a line terminator unless the `DOTALL` flag is specified.

By default, the regular expressions `^` and `$` ignore line terminators and only match at the beginning and the end, respectively, of the entire input sequence. If `MULTILINE` mode is activated then `^` matches at the beginning of input and after any line terminator except at the end of input. When in `MULTILINE` mode `$` matches just before a line terminator or the end of the input sequence.

Groups and capturing

Group number

Capturing groups are numbered by counting their opening parentheses from left to right. In the expression `((A)(B(C)))`, for example, there are four such groups:

1. `((A)(B(C)))`
2. `(A)`
3. `(B(C))`
4. `(C)`

Group zero always stands for the entire expression.

Capturing groups are so named because, during a match, each subsequence of the input sequence that matches such a group is saved. The captured subsequence may be used later in the expression, via a back reference, and may also be retrieved from the matcher once the match operation is complete.

Group name

A capturing group can also be assigned a "name", a `named-capturing group`, and then be back-referenced later by the "name". Group names are composed of the following characters. The first character must be a `letter`.

- The uppercase letters `'A'` through `'Z'` (`'\u0041'` through `'\u005a'`),
- The lowercase letters `'a'` through `'z'` (`'\u0061'` through `'\u007a'`),
- The digits `'0'` through `'9'` (`'\u0030'` through `'\u0039'`),

A `named-capturing group` is still numbered as described in Group number.

The captured input associated with a group is always the subsequence that the group most recently matched. If a group is evaluated a second time because of quantification then its previously-captured value, if any, will be retained if the second evaluation fails. Matching the string `"aba"` against the expression `(a(b)?)+`, for example, leaves group two set to `"b"`. All captured input is discarded at the beginning of each match.

Groups beginning with `(?` are either pure, *non-capturing* groups that do not capture text and do not count towards the group total, or *named-capturing* group.

Unicode support

This class is in conformance with Level 1 of *Unicode Technical Standard #18: Unicode Regular Expressions* , plus RL2.1 Canonical Equivalents and RL2.2 Extended Grapheme Clusters.

Unicode escape sequences such as `\u2014` in Java source code are processed as described in section 3.3 of *The Java Language Specification*. Such escape sequences are also implemented directly by the regular-expression parser so that Unicode escapes can be used in expressions that are read from files or from the keyboard. Thus the strings `"\u2014"` and `"\\u2014"`, while not equal, compile into the same pattern, which matches the character with hexadecimal value `0x2014`.

A Unicode character can also be represented by using its **Hex notation** (hexadecimal code point value) directly as described in construct `\x{...}`, for example a supplementary character U+2011F can be specified as `\x{2011F}`, instead of two consecutive Unicode escape sequences of the surrogate pair `\uD840\uDD1F`.

Unicode character names are supported by the named character construct `\N{...}`, for example, `\N{WHITE SMILING FACE}` specifies character `\u263A`. The character names supported by this class are the valid Unicode character names matched by `Character.codePointOf(name)`.

Unicode extended grapheme clusters are supported by the grapheme cluster matcher `\X` and the corresponding boundary matcher `\b{g}`.

Unicode scripts, blocks, categories and binary properties are written with the `\p` and `\P` constructs as in Perl. `\p{`*prop*`}` matches if the input has the property *prop*, while `\P{`*prop*`}` does not match if the input has that property.

Scripts, blocks, categories and binary properties can be used both inside and outside of a character class.

Scripts are specified either with the prefix `Is`, as in `IsHiragana`, or by using the `script` keyword (or its short form `sc`) as in `script=Hiragana` or `sc=Hiragana`.

The script names supported by `Pattern` are the valid script names accepted and defined by `UnicodeScript.forName`.

Blocks are specified with the prefix `In`, as in `InMongolian`, or by using the keyword `block` (or its short form `blk`) as in `block=Mongolian` or `blk=Mongolian`.

The block names supported by `Pattern` are the valid block names accepted and defined by `UnicodeBlock.forName`.

Categories may be specified with the optional prefix `Is`: Both `\p{L}` and `\p{IsL}` denote the category of Unicode letters. Same as scripts and blocks, categories can also be specified by using the keyword `general_category` (or its short form `gc`) as in `general_category=Lu` or `gc=Lu`.

The supported categories are those of *The Unicode Standard* in the version specified by the `Character` class. The category names are those defined in the Standard, both normative and informative.

Binary properties are specified with the prefix `Is`, as in `IsAlphabetic`. The supported binary properties by `Pattern` are

- Alphabetic
- Ideographic
- Letter
- Lowercase
- Uppercase
- Titlecase

- Punctuation
- Control
- White_Space
- Digit
- Hex_Digit
- Join_Control
- Noncharacter_Code_Point
- Assigned

The following **Predefined Character classes** and **POSIX character classes** are in conformance with the recommendation of *Annex C: Compatibility Properties* of *Unicode Technical Standard #18: Unicode Regular Expressions* , when `UNICODE_CHARACTER_CLASS` flag is specified.

Classes	Matches
`\p{Lower}`	A lowercase character:`\p{IsLowercase}`
`\p{Upper}`	An uppercase character:`\p{IsUppercase}`
`\p{ASCII}`	All ASCII:`[\x00-\x7F]`
`\p{Alpha}`	An alphabetic character:`\p{IsAlphabetic}`
`\p{Digit}`	A decimal digit character:`\p{IsDigit}`
`\p{Alnum}`	An alphanumeric character:`[\p{IsAlphabetic}\p{IsDigit}]`
`\p{Punct}`	A punctuation character:`\p{IsPunctuation}`
`\p{Graph}`	A visible character: `[^\p{IsWhite_Space}\p{gc=Cc}\p{gc=Cs}\p{gc=Cn}]`
`\p{Print}`	A printable character: `[\p{Graph}\p{Blank}&&[^\p{Cntrl}]]`
`\p{Blank}`	A space or a tab: `[\p{IsWhite_Space}&&[^\p{gc=Zl}\p{gc=Zp}\x0a\x0b\x0c\x0d\x85]]`
`\p{Cntrl}`	A control character: `\p{gc=Cc}`
`\p{XDigit}`	A hexadecimal digit: `[\p{gc=Nd}\p{IsHex_Digit}]`
`\p{Space}`	A whitespace character:`\p{IsWhite_Space}`
`\d`	A digit: `\p{IsDigit}`
`\D`	A non-digit: `[^\d]`
`\s`	A whitespace character: `\p{IsWhite_Space}`
`\S`	A non-whitespace character: `[^\s]`
`\w`	A word character: `[\p{Alpha}\p{gc=Mn}\p{gc=Me}\p{gc=Mc}\p{Digit}\p{gc=Pc}\p{IsJoin_Control}]`
`\W`	A non-word character: `[^\w]`

Categories that behave like the java.lang.Character boolean is*methodname* methods (except for the deprecated ones) are available through the same `\p{`*prop*`}` syntax where the specified property has the name `java`*`methodname`*.

Comparison to Perl 5

The `Pattern` engine performs traditional NFA-based matching with ordered alternation as occurs in Perl 5.

Perl constructs not supported by this class:

- The backreference constructs, `\g{`*n*`}` for the *n*th capturing group and `\g{`*name*`}` for named-capturing group.
- The conditional constructs `(?(`*condition*`)`*X*`)` and `(?(`*condition*`)`*X*`|`*Y*`)`,
- The embedded code constructs `(?{`*code*`})` and `(??{`*code*`})`,
- The embedded comment syntax `(?#comment)`, and
- The preprocessing operations `\l` `\u`, `\L`, and `\U`.

Constructs supported by this class but not by Perl:

- Character-class union and intersection as described above.

Notable differences from Perl:

- In Perl, `\1` through `\9` are always interpreted as back references; a backslash-escaped number greater than `9` is treated as a back reference if at least that many subexpressions exist, otherwise it is interpreted, if possible, as an octal escape. In this class octal escapes must always begin with a zero. In this class, `\1` through `\9` are always interpreted as back references, and a larger number is accepted as a back reference if at least that many subexpressions exist at that point in the regular expression, otherwise the parser will drop digits until the number is smaller or equal to the existing number of groups or it is one digit.
- Perl uses the `g` flag to request a match that resumes where the last match left off. This functionality is provided implicitly by the `Matcher` class: Repeated invocations of the `find` method will resume where the last match left off, unless the matcher is reset.
- In Perl, embedded flags at the top level of an expression affect the whole expression. In this class, embedded flags always take effect at the point at which they appear, whether they are at the top level or within a group; in the latter case, flags are restored at the end of the group just as in Perl.

For a more precise description of the behavior of regular expression constructs, please see *Mastering Regular Expressions, 3rd Edition*, Jeffrey E. F. Friedl, O'Reilly and Associates, 2006.

Since:

1.4

See Also:

`String.split(String, int)`,
`String.split(String)`,
Serialized Form

Field Summary

Fields

Modifier and Type	Field	Description
`static final int`	`CANON_EQ`	Enables canonical equivalence.
`static final int`	`CASE_INSENSITIVE`	Enables case-insensitive matching.
`static final int`	`COMMENTS`	Permits whitespace and comments in pattern.
`static final int`	`DOTALL`	Enables dotall mode.
`static final int`	`LITERAL`	Enables literal parsing of the pattern.
`static final int`	`MULTILINE`	Enables multiline mode.
`static final int`	`UNICODE_CASE`	Enables Unicode-aware case folding.
`static final int`	`UNICODE_CHARACTER_CLASS`	Enables the Unicode version of *Predefined character classes* and *POSIX character classes*.
`static final int`	`UNIX_LINES`	Enables Unix lines mode.

Method Summary

All Methods **Static Methods** **Instance Methods** **Concrete Methods**

Modifier and Type	Method	Description
`Predicate<String>`	`asMatchPredicate()`	Creates a predicate that tests if this pattern matches a given input string.
`Predicate<String>`	`asPredicate()`	Creates a predicate that tests if this pattern is found in a given input string.
`static Pattern`	`compile(String regex)`	Compiles the given regular expression into a pattern.
`static Pattern`	`compile(String regex, int flags)`	Compiles the given regular expression into a pattern with the given flags.
`int`	`flags()`	Returns this pattern's match flags.

`Matcher`	`matcher(CharSequence input)`	Creates a matcher that will match the given input against this pattern.
`static boolean`	`matches(String regex, CharSequence input)`	Compiles the given regular expression and attempts to match the given input against it.
`String`	`pattern()`	Returns the regular expression from which this pattern was compiled.
`static String`	`quote(String s)`	Returns a literal pattern `String` for the specified `String`.
`String[]`	`split(CharSequence input)`	Splits the given input sequence around matches of this pattern.
`String[]`	`split(CharSequence input, int limit)`	Splits the given input sequence around matches of this pattern.
`Stream<String>`	`splitAsStream (CharSequence input)`	Creates a stream from the given input sequence around matches of this pattern.
`String`	`toString()`	Returns the string representation of this pattern.

Methods declared in class java.lang.Object

`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `wait`, `wait`, `wait`

Field Details

UNIX_LINES

`public static final int UNIX_LINES`

Enables Unix lines mode.

In this mode, only the `'\n'` line terminator is recognized in the behavior of `.`, `^`, and `$`.

Unix lines mode can also be enabled via the embedded flag expression `(?d)`.

See Also:

Constant Field Values

CASE_INSENSITIVE

```
public static final int CASE_INSENSITIVE
```

Enables case-insensitive matching.

By default, case-insensitive matching assumes that only characters in the US-ASCII charset are being matched. Unicode-aware case-insensitive matching can be enabled by specifying the `UNICODE_CASE` flag in conjunction with this flag.

Case-insensitive matching can also be enabled via the embedded flag expression `(?i)`.

Specifying this flag may impose a slight performance penalty.

See Also:

Constant Field Values

COMMENTS

```
public static final int COMMENTS
```

Permits whitespace and comments in pattern.

In this mode, whitespace is ignored, and embedded comments starting with # are ignored until the end of a line.

Comments mode can also be enabled via the embedded flag expression `(?x)`.

See Also:

Constant Field Values

MULTILINE

```
public static final int MULTILINE
```

Enables multiline mode.

In multiline mode the expressions ^ and $ match just after or just before, respectively, a line terminator or the end of the input sequence. By default these expressions only match at the beginning and the end of the entire input sequence.

Multiline mode can also be enabled via the embedded flag expression `(?m)`.

See Also:

Constant Field Values

LITERAL

```
public static final int LITERAL
```

Enables literal parsing of the pattern.

When this flag is specified then the input string that specifies the pattern is treated as a sequence of literal characters. Metacharacters or escape sequences in the input sequence will be given no special meaning.

The flags CASE_INSENSITIVE and UNICODE_CASE retain their impact on matching when used in conjunction with this flag. The other flags become superfluous.

There is no embedded flag character for enabling literal parsing.

Since:

1.5

See Also:

Constant Field Values

DOTALL

```
public static final int DOTALL
```

Enables dotall mode.

In dotall mode, the expression `.` matches any character, including a line terminator. By default this expression does not match line terminators.

Dotall mode can also be enabled via the embedded flag expression `(?s)`. (The `s` is a mnemonic for "single-line" mode, which is what this is called in Perl.)

See Also:

Constant Field Values

UNICODE_CASE

```
public static final int UNICODE_CASE
```

Enables Unicode-aware case folding.

When this flag is specified then case-insensitive matching, when enabled by the `CASE_INSENSITIVE` flag, is done in a manner consistent with the Unicode Standard. By default, case-insensitive matching assumes that only characters in the US-ASCII charset are being matched.

Unicode-aware case folding can also be enabled via the embedded flag expression `(?u)`.

Specifying this flag may impose a performance penalty.

See Also:

Constant Field Values

CANON_EQ

```
public static final int CANON_EQ
```

Enables canonical equivalence.

When this flag is specified then two characters will be considered to match if, and only if, their full canonical decompositions match. The expression `"a\u030A"`, for example, will match the string `"\u00E5"` when this flag is specified. By default, matching does not take canonical equivalence into account.

There is no embedded flag character for enabling canonical equivalence.

Specifying this flag may impose a performance penalty.

See Also:

Constant Field Values

UNICODE_CHARACTER_CLASS

```
public static final int UNICODE_CHARACTER_CLASS
```

Enables the Unicode version of *Predefined character classes* and *POSIX character classes*.

When this flag is specified then the (US-ASCII only) *Predefined character classes* and *POSIX character classes* are in conformance with *Unicode Technical Standard #18: Unicode Regular Expressions* *Annex C: Compatibility Properties*.

The UNICODE_CHARACTER_CLASS mode can also be enabled via the embedded flag expression `(?U)`.

The flag implies UNICODE_CASE, that is, it enables Unicode-aware case folding.

Specifying this flag may impose a performance penalty.

Since:

1.7

See Also:

Constant Field Values

Method Details

compile

```
public static Pattern compile(String regex)
```

Compiles the given regular expression into a pattern.

Parameters:

`regex` - The expression to be compiled

Returns:

the given regular expression compiled into a pattern

Throws:

`PatternSyntaxException` - If the expression's syntax is invalid

compile

```
public static Pattern compile(String regex,
                              int flags)
```

Compiles the given regular expression into a pattern with the given flags.

Parameters:

`regex` - The expression to be compiled

`flags` - Match flags, a bit mask that may include `CASE_INSENSITIVE`, `MULTILINE`, `DOTALL`, `UNICODE_CASE`, `CANON_EQ`, `UNIX_LINES`, `LITERAL`, `UNICODE_CHARACTER_CLASS` and `COMMENTS`

Returns:

the given regular expression compiled into a pattern with the given flags

Throws:

`IllegalArgumentException` - If bit values other than those corresponding to the defined match flags are set in `flags`

`PatternSyntaxException` - If the expression's syntax is invalid

pattern

```
public String pattern()
```

Returns the regular expression from which this pattern was compiled.

Returns:

The source of this pattern

toString

```
public String toString()
```

Returns the string representation of this pattern. This is the regular expression from which this pattern was compiled.

Overrides:

`toString` in class `Object`

Returns:

The string representation of this pattern

Since:

1.5

matcher

```
public Matcher matcher(CharSequence input)
```

Creates a matcher that will match the given input against this pattern.

Parameters:

`input` - The character sequence to be matched

Returns:

A new matcher for this pattern

flags

```
public int flags()
```

Returns this pattern's match flags.

Returns:

The match flags specified when this pattern was compiled

matches

```
public static boolean matches(String regex,
                              CharSequence input)
```

Compiles the given regular expression and attempts to match the given input against it.

An invocation of this convenience method of the form

```
    Pattern.matches(regex, input);
```

behaves in exactly the same way as the expression

```
      Pattern.compile(regex).matcher(input).matches()
```

If a pattern is to be used multiple times, compiling it once and reusing it will be more efficient than invoking this method each time.

Parameters:

`regex` - The expression to be compiled

`input` - The character sequence to be matched

Returns:

whether or not the regular expression matches on the input

Throws:

`PatternSyntaxException` - If the expression's syntax is invalid

split

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

Splits the given input sequence around matches of this pattern.

The array returned by this method contains each substring of the input sequence that is terminated by another subsequence that matches this pattern or is terminated by the end of the input sequence. The substrings in the array are in the order in which they occur in the input. If this pattern does not match any subsequence of the input then the resulting array has just one element, namely the input sequence in string form.

When there is a positive-width match at the beginning of the input sequence then an empty leading substring is included at the beginning of the resulting array. A zero-width match at the beginning however never produces such empty leading substring.

The `limit` parameter controls the number of times the pattern is applied and therefore affects the length of the resulting array.

- If the *limit* is positive then the pattern will be applied at most *limit* - 1 times, the array's length will be no greater than *limit*, and the array's last entry will contain all input beyond the last matched delimiter.
- If the *limit* is zero then the pattern will be applied as many times as possible, the array can have any length, and trailing empty strings will be discarded.
- If the *limit* is negative then the pattern will be applied as many times as possible and the array can have any length.

The input `"boo:and:foo"`, for example, yields the following results with these parameters:

Regex	Limit	Result

Regex	Limit	Result
:	2	`{ "boo", "and:foo" }`
	5	`{ "boo", "and", "foo" }`
	-2	`{ "boo", "and", "foo" }`
o	5	`{ "b", "", ":and:f", "", "" }`
	-2	`{ "b", "", ":and:f", "", "" }`
	0	`{ "b", "", ":and:f" }`

Parameters:

`input` - The character sequence to be split

`limit` - The result threshold, as described above

Returns:

The array of strings computed by splitting the input around matches of this pattern

split

`public String[] split(CharSequence input)`

Splits the given input sequence around matches of this pattern.

This method works as if by invoking the two-argument `split` method with the given input sequence and a limit argument of zero. Trailing empty strings are therefore not included in the resulting array.

The input `"boo:and:foo"`, for example, yields the following results with these expressions:

Regex	Result
:	`{ "boo", "and", "foo" }`
o	`{ "b", "", ":and:f" }`

Parameters:

`input` - The character sequence to be split

Returns:

The array of strings computed by splitting the input around matches of this pattern

quote

`public static String quote(String s)`

Returns a literal pattern `String` for the specified `String`.

This method produces a `String` that can be used to create a `Pattern` that would match the string `s` as if it were a literal pattern.

Metacharacters or escape sequences in the input sequence will be given no special meaning.

Parameters:

`s` - The string to be literalized

Returns:

A literal string replacement

Since:

1.5

asPredicate

```
public Predicate<String> asPredicate()
```

Creates a predicate that tests if this pattern is found in a given input string.

API Note:

This method creates a predicate that behaves as if it creates a matcher from the input sequence and then calls `find`, for example a predicate of the form:

```
   s -> matcher(s).find();
```

Returns:

The predicate which can be used for finding a match on a subsequence of a string

Since:

1.8

See Also:

`Matcher.find()`

asMatchPredicate

```
public Predicate<String> asMatchPredicate()
```

Creates a predicate that tests if this pattern matches a given input string.

API Note:

This method creates a predicate that behaves as if it creates a matcher from the input sequence and then calls `matches`, for example a predicate of the form:

```
   s -> matcher(s).matches();
```

Returns:

The predicate which can be used for matching an input string against this pattern.

Since:

11

See Also:

`Matcher.matches()`

splitAsStream

```
public Stream<String> splitAsStream(CharSequence input)
```

Creates a stream from the given input sequence around matches of this pattern.

The stream returned by this method contains each substring of the input sequence that is terminated by another subsequence that matches this pattern or is terminated by the end of the input sequence. The substrings in the stream are in the order in which they occur in the input. Trailing empty strings will be discarded and not encountered in the stream.

If this pattern does not match any subsequence of the input then the resulting stream has just one element, namely the input sequence in string form.

When there is a positive-width match at the beginning of the input sequence then an empty leading substring is included at the beginning of the stream. A zero-width match at the beginning however never produces such empty leading substring.

If the input sequence is mutable, it must remain constant during the execution of the terminal stream operation. Otherwise, the result of the terminal stream operation is undefined.

Parameters:

`input` - The character sequence to be split

Returns:

The stream of strings computed by splitting the input around matches of this pattern

Since:

1.8

See Also:

`split(CharSequence)`

Report a bug or suggest an enhancement