java.util.regex

Class Matcher

java.lang.Object
    java.util.regex.Matcher

All Implemented Interfaces:

MatchResult

```
public final class Matcher
extends Object
implements MatchResult
```

An engine that performs match operations on a character sequence by interpreting a `Pattern`.

A matcher is created from a pattern by invoking the pattern's `matcher` method. Once created, a matcher can be used to perform three different kinds of match operations:

- The `matches` method attempts to match the entire input sequence against the pattern.
- The `lookingAt` method attempts to match the input sequence, starting at the beginning, against the pattern.
- The `find` method scans the input sequence looking for the next subsequence that matches the pattern.

Each of these methods returns a boolean indicating success or failure. More information about a successful match can be obtained by querying the state of the matcher.

A matcher finds matches in a subset of its input called the *region*. By default, the region contains all of the matcher's input. The region can be modified via the`region` method and queried via the `regionStart` and `regionEnd` methods. The way that the region boundaries interact with some pattern constructs can be changed. See `useAnchoringBounds` and `useTransparentBounds` for more details.

This class also defines methods for replacing matched subsequences with new strings whose contents can, if desired, be computed from the match result. The `appendReplacement` and `appendTail` methods can be used in tandem in order to collect the result into an existing string buffer, or the more convenient `replaceAll` method can be used to create a string in which every matching subsequence in the input sequence is replaced.

The explicit state of a matcher includes the start and end indices of the most recent successful match. It also includes the start and end indices of the input subsequence captured by each capturing group in the pattern as well as a total count of such subsequences. As a convenience, methods are also provided for returning these captured subsequences in string form.

The explicit state of a matcher is initially undefined; attempting to query any part of it before a successful match

will cause an `IllegalStateException` to be thrown. The explicit state of a matcher is recomputed by every match operation.

The implicit state of a matcher includes the input character sequence as well as the *append position*, which is initially zero and is updated by the `appendReplacement` method.

A matcher may be reset explicitly by invoking its `reset()` method or, if a new input sequence is desired, its `reset(CharSequence)` method. Resetting a matcher discards its explicit state information and sets the append position to zero.

Instances of this class are not safe for use by multiple concurrent threads.

Since:

1.4

Method Summary

Methods

Modifier and Type	Method and Description
`Matcher`	**`appendReplacement`**`(StringBuffer sb, String replacement)` Implements a non-terminal append-and-replace step.
`StringBuffer`	**`appendTail`**`(StringBuffer sb)` Implements a terminal append-and-replace step.
`int`	**`end`**`()` Returns the offset after the last character matched.
`int`	**`end`**`(int group)` Returns the offset after the last character of the subsequence captured by the given group during the previous match operation.
`boolean`	**`find`**`()` Attempts to find the next subsequence of the input sequence that matches the pattern.
`boolean`	**`find`**`(int start)` Resets this matcher and then attempts to find the next subsequence of the input sequence that matches the pattern, starting at the specified index.
`String`	**`group`**`()` Returns the input subsequence matched by the previous match.
`String`	**`group`**`(int group)` Returns the input subsequence captured by the given group during the previous match operation.
`String`	**`group`**`(String name)` Returns the input subsequence captured by the given **named-capturing group** during the previous match operation.

int	**groupCount**() Returns the number of capturing groups in this matcher's pattern.
boolean	**hasAnchoringBounds**() Queries the anchoring of region bounds for this matcher.
boolean	**hasTransparentBounds**() Queries the transparency of region bounds for this matcher.
boolean	**hitEnd**() Returns true if the end of input was hit by the search engine in the last match operation performed by this matcher.
boolean	**lookingAt**() Attempts to match the input sequence, starting at the beginning of the region, against the pattern.
boolean	**matches**() Attempts to match the entire region against the pattern.
Pattern	**pattern**() Returns the pattern that is interpreted by this matcher.
static **String**	**quoteReplacement**(**String** s) Returns a literal replacement `String` for the specified `String`.
Matcher	**region**(int start, int end) Sets the limits of this matcher's region.
int	**regionEnd**() Reports the end index (exclusive) of this matcher's region.
int	**regionStart**() Reports the start index of this matcher's region.
String	**replaceAll**(**String** replacement) Replaces every subsequence of the input sequence that matches the pattern with the given replacement string.
String	**replaceFirst**(**String** replacement) Replaces the first subsequence of the input sequence that matches the pattern with the given replacement string.
boolean	**requireEnd**() Returns true if more input could change a positive match into a negative one.
Matcher	**reset**() Resets this matcher.
Matcher	**reset**(**CharSequence** input) Resets this matcher with a new input sequence.
int	**start**() Returns the start index of the previous match.

int	**start**(int group) Returns the start index of the subsequence captured by the given group during the previous match operation.
MatchResult	**toMatchResult**() Returns the match state of this matcher as a **MatchResult**.
String	**toString**() Returns the string representation of this matcher.
Matcher	**useAnchoringBounds**(boolean b) Sets the anchoring of region bounds for this matcher.
Matcher	**usePattern**(**Pattern** newPattern) Changes the Pattern that this Matcher uses to find matches with.
Matcher	**useTransparentBounds**(boolean b) Sets the transparency of region bounds for this matcher.

Methods inherited from class java.lang.Object

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

Method Detail

pattern

```
public Pattern pattern()
```

Returns the pattern that is interpreted by this matcher.

Returns:

The pattern for which this matcher was created

toMatchResult

```
public MatchResult toMatchResult()
```

Returns the match state of this matcher as a MatchResult. The result is unaffected by subsequent operations performed upon this matcher.

Returns:

a MatchResult with the state of this matcher

Since:

1.5

usePattern

```
public Matcher usePattern(Pattern newPattern)
```

Changes the `Pattern` that this `Matcher` uses to find matches with.

This method causes this matcher to lose information about the groups of the last match that occurred. The matcher's position in the input is maintained and its last append position is unaffected.

Parameters:

`newPattern` - The new pattern used by this matcher

Returns:

This matcher

Throws:

`IllegalArgumentException` - If newPattern is `null`

Since:

1.5

reset

```
public Matcher reset()
```

Resets this matcher.

Resetting a matcher discards all of its explicit state information and sets its append position to zero. The matcher's region is set to the default region, which is its entire character sequence. The anchoring and transparency of this matcher's region boundaries are unaffected.

Returns:

This matcher

reset

```
public Matcher reset(CharSequence input)
```

Resets this matcher with a new input sequence.

Resetting a matcher discards all of its explicit state information and sets its append position to zero. The matcher's region is set to the default region, which is its entire character sequence. The anchoring and transparency of this matcher's region boundaries are unaffected.

Parameters:

`input` - The new input character sequence

Returns:

This matcher

start

```
public int start()
```

Returns the start index of the previous match.

Specified by:

`start` in interface `MatchResult`

Returns:

The index of the first character matched

Throws:

`IllegalStateException` - If no match has yet been attempted, or if the previous match operation failed

start

```
public int start(int group)
```

Returns the start index of the subsequence captured by the given group during the previous match operation.

Capturing groups are indexed from left to right, starting at one. Group zero denotes the entire pattern, so the expression *m.*`start(0)` is equivalent to *m.*`start()`.

Specified by:

`start` in interface `MatchResult`

Parameters:

`group` - The index of a capturing group in this matcher's pattern

Returns:

The index of the first character captured by the group, or `-1` if the match was successful but the

group itself did not match anything

Throws:

`IllegalStateException` - If no match has yet been attempted, or if the previous match operation failed

`IndexOutOfBoundsException` - If there is no capturing group in the pattern with the given index

end

```
public int end()
```

Returns the offset after the last character matched.

Specified by:

`end` in interface `MatchResult`

Returns:

The offset after the last character matched

Throws:

`IllegalStateException` - If no match has yet been attempted, or if the previous match operation failed

end

```
public int end(int group)
```

Returns the offset after the last character of the subsequence captured by the given group during the previous match operation.

Capturing groups are indexed from left to right, starting at one. Group zero denotes the entire pattern, so the expression *m.*`end(0)` is equivalent to *m.*`end()`.

Specified by:

`end` in interface `MatchResult`

Parameters:

`group` - The index of a capturing group in this matcher's pattern

Returns:

The offset after the last character captured by the group, or `-1` if the match was successful but the group itself did not match anything

Throws:

`IllegalStateException` - If no match has yet been attempted, or if the previous match operation failed

`IndexOutOfBoundsException` - If there is no capturing group in the pattern with the given index

group

```
public String group()
```

Returns the input subsequence matched by the previous match.

For a matcher *m* with input sequence *s*, the expressions *m.*`group()` and *s.*`substring(`*m.*`start(),` *m.*`end())` are equivalent.

Note that some patterns, for example `a*`, match the empty string. This method will return the empty string when the pattern successfully matches the empty string in the input.

Specified by:

`group` in interface `MatchResult`

Returns:

The (possibly empty) subsequence matched by the previous match, in string form

Throws:

`IllegalStateException` - If no match has yet been attempted, or if the previous match operation failed

group

```
public String group(int group)
```

Returns the input subsequence captured by the given group during the previous match operation.

For a matcher *m*, input sequence *s*, and group index *g*, the expressions *m.*`group(`*g*`)` and *s.*`substring(`*m.*`start(`*g*`),` *m.*`end(`*g*`))` are equivalent.

Capturing groups are indexed from left to right, starting at one. Group zero denotes the entire pattern, so the expression `m.group(0)` is equivalent to `m.group()`.

If the match was successful but the group specified failed to match any part of the input sequence, then `null` is returned. Note that some groups, for example `(a*)`, match the empty string. This method will return the empty string when such a group successfully matches the empty string in the input.

Specified by:

group in interface MatchResult

Parameters:

group - The index of a capturing group in this matcher's pattern

Returns:

The (possibly empty) subsequence captured by the group during the previous match, or null if the group failed to match part of the input

Throws:

IllegalStateException - If no match has yet been attempted, or if the previous match operation failed

IndexOutOfBoundsException - If there is no capturing group in the pattern with the given index

group

```
public String group(String name)
```

Returns the input subsequence captured by the given named-capturing group during the previous match operation.

If the match was successful but the group specified failed to match any part of the input sequence, then null is returned. Note that some groups, for example (a*), match the empty string. This method will return the empty string when such a group successfully matches the empty string in the input.

Parameters:

name - The name of a named-capturing group in this matcher's pattern

Returns:

The (possibly empty) subsequence captured by the named group during the previous match, or null if the group failed to match part of the input

Throws:

IllegalStateException - If no match has yet been attempted, or if the previous match operation failed

IllegalArgumentException - If there is no capturing group in the pattern with the given name

Since:

1.7

groupCount

```
public int groupCount()
```

Returns the number of capturing groups in this matcher's pattern.

Group zero denotes the entire pattern by convention. It is not included in this count.

Any non-negative integer smaller than or equal to the value returned by this method is guaranteed to be a valid group index for this matcher.

Specified by:

`groupCount` in interface `MatchResult`

Returns:

The number of capturing groups in this matcher's pattern

matches

```
public boolean matches()
```

Attempts to match the entire region against the pattern.

If the match succeeds then more information can be obtained via the `start`, `end`, and `group` methods.

Returns:

`true` if, and only if, the entire region sequence matches this matcher's pattern

find

```
public boolean find()
```

Attempts to find the next subsequence of the input sequence that matches the pattern.

This method starts at the beginning of this matcher's region, or, if a previous invocation of the method was successful and the matcher has not since been reset, at the first character not matched by the previous match.

If the match succeeds then more information can be obtained via the `start`, `end`, and `group` methods.

Returns:

`true` if, and only if, a subsequence of the input sequence matches this matcher's pattern

find

```
public boolean find(int start)
```

Resets this matcher and then attempts to find the next subsequence of the input sequence that matches the pattern, starting at the specified index.

If the match succeeds then more information can be obtained via the `start`, `end`, and `group` methods, and subsequent invocations of the `find()` method will start at the first character not matched by this match.

Returns:

`true` if, and only if, a subsequence of the input sequence starting at the given index matches this matcher's pattern

Throws:

`IndexOutOfBoundsException` - If start is less than zero or if start is greater than the length of the input sequence.

lookingAt

```
public boolean lookingAt()
```

Attempts to match the input sequence, starting at the beginning of the region, against the pattern.

Like the `matches` method, this method always starts at the beginning of the region; unlike that method, it does not require that the entire region be matched.

If the match succeeds then more information can be obtained via the `start`, `end`, and `group` methods.

Returns:

`true` if, and only if, a prefix of the input sequence matches this matcher's pattern

quoteReplacement

```
public static String quoteReplacement(String s)
```

Returns a literal replacement `String` for the specified `String`. This method produces a `String` that will work as a literal replacement `s` in the `appendReplacement` method of the `Matcher` class. The `String` produced will match the sequence of characters in `s` treated as a literal sequence. Slashes ('\') and dollar signs ('$') will be given no special meaning.

Parameters:

`s` - The string to be literalized

Returns:

A literal string replacement

Since:

1.5

appendReplacement

```
public Matcher appendReplacement(StringBuffer sb,
                                 String replacement)
```

Implements a non-terminal append-and-replace step.

This method performs the following actions:

1. It reads characters from the input sequence, starting at the append position, and appends them to the given string buffer. It stops after reading the last character preceding the previous match, that is, the character at index `start() - 1`.
2. It appends the given replacement string to the string buffer.
3. It sets the append position of this matcher to the index of the last character matched, plus one, that is, to `end()`.

The replacement string may contain references to subsequences captured during the previous match: Each occurrence of `${`*name*`}` or `$`*g* will be replaced by the result of evaluating the corresponding `group(name)` or `group(g)` respectively. For `$`*g*, the first number after the `$` is always treated as part of the group reference. Subsequent numbers are incorporated into g if they would form a legal group reference. Only the numerals '0' through '9' are considered as potential components of the group reference. If the second group matched the string `"foo"`, for example, then passing the replacement string `"$2bar"` would cause `"foobar"` to be appended to the string buffer. A dollar sign (`$`) may be included as a literal in the replacement string by preceding it with a backslash (`\$`).

Note that backslashes (`\`) and dollar signs (`$`) in the replacement string may cause the results to be different than if it were being treated as a literal replacement string. Dollar signs may be treated as references to captured subsequences as described above, and backslashes are used to escape literal characters in the replacement string.

This method is intended to be used in a loop together with the `appendTail` and `find` methods. The following code, for example, writes `one dog two dogs in the yard` to the standard-output stream:

```
Pattern p = Pattern.compile("cat");
Matcher m = p.matcher("one cat two cats in the yard");
StringBuffer sb = new StringBuffer();
while (m.find()) {
    m.appendReplacement(sb, "dog");
}
m.appendTail(sb);
System.out.println(sb.toString());
```

Parameters:

`sb` - The target string buffer

`replacement` - The replacement string

Returns:

This matcher

Throws:

`IllegalStateException` - If no match has yet been attempted, or if the previous match operation failed

`IllegalArgumentException` - If the replacement string refers to a named-capturing group that does not exist in the pattern

`IndexOutOfBoundsException` - If the replacement string refers to a capturing group that does not exist in the pattern

appendTail

```
public StringBuffer appendTail(StringBuffer sb)
```

Implements a terminal append-and-replace step.

This method reads characters from the input sequence, starting at the append position, and appends them to the given string buffer. It is intended to be invoked after one or more invocations of the `appendReplacement` method in order to copy the remainder of the input sequence.

Parameters:

`sb` - The target string buffer

Returns:

The target string buffer

replaceAll

```
public String replaceAll(String replacement)
```

Replaces every subsequence of the input sequence that matches the pattern with the given replacement string.

This method first resets this matcher. It then scans the input sequence looking for matches of the pattern. Characters that are not part of any match are appended directly to the result string; each match is replaced in the result by the replacement string. The replacement string may contain references to captured subsequences as in the `appendReplacement` method.

Note that backslashes (\) and dollar signs ($) in the replacement string may cause the results to be

different than if it were being treated as a literal replacement string. Dollar signs may be treated as references to captured subsequences as described above, and backslashes are used to escape literal characters in the replacement string.

Given the regular expression `a*b`, the input `"aabfooaabfooabfoob"`, and the replacement string `"-"`, an invocation of this method on a matcher for that expression would yield the string `"-foo-foo-foo-"`.

Invoking this method changes this matcher's state. If the matcher is to be used in further matching operations then it should first be reset.

Parameters:

`replacement` - The replacement string

Returns:

The string constructed by replacing each matching subsequence by the replacement string, substituting captured subsequences as needed

replaceFirst

```
public String replaceFirst(String replacement)
```

Replaces the first subsequence of the input sequence that matches the pattern with the given replacement string.

This method first resets this matcher. It then scans the input sequence looking for a match of the pattern. Characters that are not part of the match are appended directly to the result string; the match is replaced in the result by the replacement string. The replacement string may contain references to captured subsequences as in the `appendReplacement` method.

Note that backslashes (`\`) and dollar signs (`$`) in the replacement string may cause the results to be different than if it were being treated as a literal replacement string. Dollar signs may be treated as references to captured subsequences as described above, and backslashes are used to escape literal characters in the replacement string.

Given the regular expression `dog`, the input `"zzzdogzzzdogzzz"`, and the replacement string `"cat"`, an invocation of this method on a matcher for that expression would yield the string `"zzzcatzzzdogzzz"`.

Invoking this method changes this matcher's state. If the matcher is to be used in further matching operations then it should first be reset.

Parameters:

`replacement` - The replacement string

Returns:

The string constructed by replacing the first matching subsequence by the replacement string, substituting captured subsequences as needed

region

```
public Matcher region(int start,
                      int end)
```

Sets the limits of this matcher's region. The region is the part of the input sequence that will be searched to find a match. Invoking this method resets the matcher, and then sets the region to start at the index specified by the `start` parameter and end at the index specified by the `end` parameter.

Depending on the transparency and anchoring being used (see `useTransparentBounds` and `useAnchoringBounds`), certain constructs such as anchors may behave differently at or around the boundaries of the region.

Parameters:

`start` - The index to start searching at (inclusive)

`end` - The index to end searching at (exclusive)

Returns:

this matcher

Throws:

`IndexOutOfBoundsException` - If start or end is less than zero, if start is greater than the length of the input sequence, if end is greater than the length of the input sequence, or if start is greater than end.

Since:

1.5

regionStart

```
public int regionStart()
```

Reports the start index of this matcher's region. The searches this matcher conducts are limited to finding matches within `regionStart` (inclusive) and `regionEnd` (exclusive).

Returns:

The starting point of this matcher's region

Since:

1.5

regionEnd

public int regionEnd()

Reports the end index (exclusive) of this matcher's region. The searches this matcher conducts are limited to finding matches within `regionStart` (inclusive) and `regionEnd` (exclusive).

Returns:

the ending point of this matcher's region

Since:

1.5

hasTransparentBounds

public boolean hasTransparentBounds()

Queries the transparency of region bounds for this matcher.

This method returns `true` if this matcher uses *transparent* bounds, `false` if it uses *opaque* bounds.

See `useTransparentBounds` for a description of transparent and opaque bounds.

By default, a matcher uses opaque region boundaries.

Returns:

`true` iff this matcher is using transparent bounds, `false` otherwise.

Since:

1.5

See Also:

`useTransparentBounds(boolean)`

useTransparentBounds

public Matcher useTransparentBounds(boolean b)

Sets the transparency of region bounds for this matcher.

Invoking this method with an argument of `true` will set this matcher to use *transparent* bounds. If the boolean argument is `false`, then *opaque* bounds will be used.

Using transparent bounds, the boundaries of this matcher's region are transparent to lookahead, lookbehind, and boundary matching constructs. Those constructs can see beyond the boundaries of the region to see if a match is appropriate.

Using opaque bounds, the boundaries of this matcher's region are opaque to lookahead, lookbehind, and boundary matching constructs that may try to see beyond them. Those constructs cannot look past

the boundaries so they will fail to match anything outside of the region.

By default, a matcher uses opaque bounds.

Parameters:

`b` - a boolean indicating whether to use opaque or transparent regions

Returns:

this matcher

Since:

1.5

See Also:

`hasTransparentBounds()`

hasAnchoringBounds

```
public boolean hasAnchoringBounds()
```

Queries the anchoring of region bounds for this matcher.

This method returns `true` if this matcher uses *anchoring* bounds, `false` otherwise.

See `useAnchoringBounds` for a description of anchoring bounds.

By default, a matcher uses anchoring region boundaries.

Returns:

`true` iff this matcher is using anchoring bounds, `false` otherwise.

Since:

1.5

See Also:

`useAnchoringBounds(boolean)`

useAnchoringBounds

```
public Matcher useAnchoringBounds(boolean b)
```

Sets the anchoring of region bounds for this matcher.

Invoking this method with an argument of `true` will set this matcher to use *anchoring* bounds. If the boolean argument is `false`, then *non-anchoring* bounds will be used.

Using anchoring bounds, the boundaries of this matcher's region match anchors such as ^ and $.

Without anchoring bounds, the boundaries of this matcher's region will not match anchors such as ^ and $.

By default, a matcher uses anchoring region boundaries.

Parameters:

b - a boolean indicating whether or not to use anchoring bounds.

Returns:

this matcher

Since:

1.5

See Also:

`hasAnchoringBounds()`

toString

```
public String toString()
```

Returns the string representation of this matcher. The string representation of a `Matcher` contains information that may be useful for debugging. The exact format is unspecified.

Overrides:

`toString` in class `Object`

Returns:

The string representation of this matcher

Since:

1.5

hitEnd

```
public boolean hitEnd()
```

Returns true if the end of input was hit by the search engine in the last match operation performed by this matcher.

When this method returns true, then it is possible that more input would have changed the result of the last search.

Returns:

true iff the end of input was hit in the last match; false otherwise

Since:

1.5

requireEnd

```
public boolean requireEnd()
```

Returns true if more input could change a positive match into a negative one.

If this method returns true, and a match was found, then more input could cause the match to be lost. If this method returns false and a match was found, then more input might change the match but the match won't be lost. If a match was not found, then requireEnd has no meaning.

Returns:

true iff more input could change a positive match into a negative one.

Since:

1.5