Expresiones Regulares en Java

Traductores

Guillermo Licea Sandoval

What Are Regular Expressions?

Regular expressions are a way to describe a set of strings based on common characteristics shared by each string in the set. They can be used to search, edit, or manipulate text and data. You must learn a specific syntax to create regular expressions — one that goes beyond the normal syntax of the Java programming language. Regular expressions vary in complexity, but once you understand the basics of how they're constructed, you'll be able to decipher (or create) any regular expression.

This trail teaches the regular expression syntax supported by the `java.util.regex` API and presents several working examples to illustrate how the various objects interact. In the world of regular expressions, there are many different flavors to choose from, such as grep, Perl, Tcl, Python, PHP, and awk. The regular expression syntax in the `java.util.regex` API is most similar to that found in Perl.

How Are Regular Expressions Represented in This Package?

The `java.util.regex` package primarily consists of three classes: `Pattern`, `Matcher`, and `PatternSyntaxException`.

- A `Pattern` object is a compiled representation of a regular expression. The `Pattern` class provides no public constructors. To create a pattern, you must first invoke one of its `public static compile` methods, which will then return a `Pattern` object. These methods accept a regular expression as the first argument; the first few lessons of this trail will teach you the required syntax.
- A `Matcher` object is the engine that interprets the pattern and performs match operations against an input string. Like the `Pattern` class, `Matcher` defines no public constructors. You obtain a `Matcher` object by invoking the `matcher` method on a `Pattern` object.
- A `PatternSyntaxException` object is an unchecked exception that indicates a syntax error in a regular expression pattern.

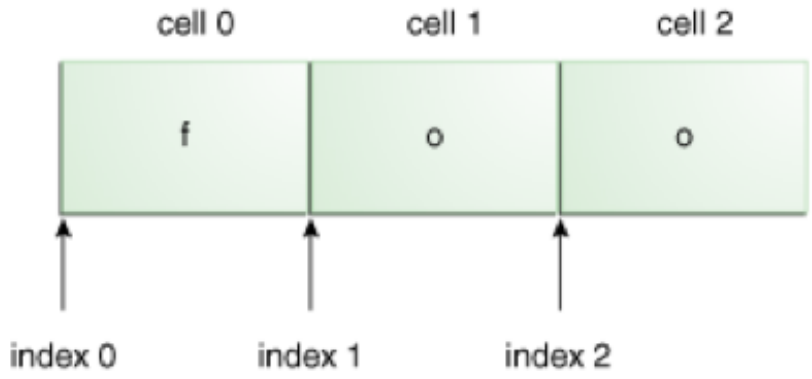
```
import java.io.Console;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexTestHarness {

    public static void main(String[] args){
        Console console = System.console();
        if (console == null) {
            System.err.println("No console.");
            System.exit(1);
        }
        while (true) {

            Pattern pattern =
            Pattern.compile(console.readLine("%nEnter your regex: "));

            Matcher matcher =
            pattern.matcher(console.readLine("Enter input string to search: "));

            boolean found = false;
            while (matcher.find()) {
                console.format("I found the text" +
                    " \"%s\" starting at " +
                    "index %d and ending at index %d.%n",
                    matcher.group(),
                    matcher.start(),
                    matcher.end());
                found = true;
            }
            if(!found){
                console.format("No match found.%n");
            }
        }
    }
}
```

String Literals

The most basic form of pattern matching supported by this API is the match of a string literal. For example, if the regular expression is `foo` and the input string is `foo`, the match will succeed because the strings are identical. Try this out with the test harness:

```
Enter your regex: foo
Enter input string to search: foo
I found the text foo starting at index 0 and ending at index 3.
```

This match was a success. Note that while the input string is 3 characters long, the start index is 0 and the end index is 3. By convention, ranges are inclusive of the beginning index and exclusive of the end index, as shown in the following figure:



The string literal foo, with numbered cells and index values.

Metacharacters

This API also supports a number of special characters that affect the way a pattern is matched. Change the regular expression to `cat.` and the input string to `cats`. The output will appear as follows:

```
Enter your regex: cat.
Enter input string to search: cats
I found the text cats starting at index 0 and ending at index 4.
```

The match still succeeds, even though the dot "`.`" is not present in the input string. It succeeds because the dot is a *metacharacter* — a character with special meaning interpreted by the matcher. The metacharacter "." means "any character" which is why the match succeeds in this example.

The metacharacters supported by this API are: `<([{\^-=$!|]})?*+.>`

There are two ways to force a metacharacter to be treated as an ordinary character:

- precede the metacharacter with a backslash, or
- enclose it within `\Q` (which starts the quote) and `\E` (which ends it).

When using this technique, the `\Q` and `\E` can be placed at any location within the expression, provided that the `\Q` comes first.

Character Classes

If you browse through the `Pattern` class specification, you'll see tables summarizing the supported regular expression constructs. In the "Character Classes" section you'll find the following:

Construct	Description
`[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)

The left-hand column specifies the regular expression constructs, while the right-hand column describes the conditions under which each construct will match.

Simple Classes

The most basic form of a character class is to simply place a set of characters side-by-side within square brackets. For example, the regular expression `[bcr]at` will match the words "bat", "cat", or "rat" because it defines a character class (accepting either "b", "c", or "r") as its first character.

```
Enter your regex: [bcr]at
Enter input string to search: bat
I found the text "bat" starting at index 0 and ending at index 3.

Enter your regex: [bcr]at
Enter input string to search: cat
I found the text "cat" starting at index 0 and ending at index 3.

Enter your regex: [bcr]at
Enter input string to search: rat
I found the text "rat" starting at index 0 and ending at index 3.

Enter your regex: [bcr]at
Enter input string to search: hat
No match found.
```

Negation

To match all characters *except* those listed, insert the "^" metacharacter at the beginning of the character class. This technique is known as *negation*.

```
Enter your regex: [^bcr]at
Enter input string to search: bat
No match found.

Enter your regex: [^bcr]at
Enter input string to search: cat
No match found.

Enter your regex: [^bcr]at
Enter input string to search: rat
No match found.

Enter your regex: [^bcr]at
Enter input string to search: hat
I found the text "hat" starting at index 0 and ending at index 3.
```

The match is successful only if the first character of the input string does *not* contain any of the characters defined by the character class.

Ranges

Sometimes you'll want to define a character class that includes a range of values, such as the letters "a through h" or the numbers "1 through 5". To specify a range, simply insert the "-" metacharacter between the first and last character to be matched, such as `[1-5]` or `[a-h]`. You can also place different ranges beside each other within the class to further expand the match possibilities. For example, `[a-zA-Z]` will match any letter of the alphabet: a to z (lowercase) or A to Z (uppercase).

Here are some examples of ranges and negation:

```
Enter your regex: [a-c]
Enter input string to search: a
I found the text "a" starting at index 0 and ending at index 1.

Enter your regex: [a-c]
Enter input string to search: b
I found the text "b" starting at index 0 and ending at index 1.

Enter your regex: [a-c]
Enter input string to search: c
I found the text "c" starting at index 0 and ending at index 1.

Enter your regex: [a-c]
Enter input string to search: d
No match found.

Enter your regex: foo[1-5]
Enter input string to search: foo1
I found the text "foo1" starting at index 0 and ending at index 4.
```

Unions

You can also use *unions* to create a single character class comprised of two or more separate character classes. To create a union, simply nest one class inside the other, such as `[0-4[6-8]]`. This particular union creates a single character class that matches the numbers 0, 1, 2, 3, 4, 6, 7, and 8.

```
Enter your regex: [0-4[6-8]]
Enter input string to search: 0
I found the text "0" starting at index 0 and ending at index 1.

Enter your regex: [0-4[6-8]]
Enter input string to search: 5
No match found.

Enter your regex: [0-4[6-8]]
Enter input string to search: 6
I found the text "6" starting at index 0 and ending at index 1.

Enter your regex: [0-4[6-8]]
Enter input string to search: 8
I found the text "8" starting at index 0 and ending at index 1.

Enter your regex: [0-4[6-8]]
Enter input string to search: 9
No match found.
```

Intersections

To create a single character class matching only the characters common to all of its nested classes, use &&, as in `[0-9&&[345]]`. This particular intersection creates a single character class matching only the numbers common to both character classes: 3, 4, and 5.

```
Enter your regex: [0-9&&[345]]
Enter input string to search: 3
I found the text "3" starting at index 0 and ending at index 1.

Enter your regex: [0-9&&[345]]
Enter input string to search: 4
I found the text "4" starting at index 0 and ending at index 1.

Enter your regex: [0-9&&[345]]
Enter input string to search: 5
I found the text "5" starting at index 0 and ending at index 1.

Enter your regex: [0-9&&[345]]
Enter input string to search: 2
No match found.

Enter your regex: [0-9&&[345]]
Enter input string to search: 6
No match found.
```

And here's an example that shows the intersection of two ranges:

```
Enter your regex: [2-8&&[4-6]]
Enter input string to search: 3
No match found.

Enter your regex: [2-8&&[4-6]]
Enter input string to search: 4
I found the text "4" starting at index 0 and ending at index 1.

Enter your regex: [2-8&&[4-6]]
Enter input string to search: 5
I found the text "5" starting at index 0 and ending at index 1.

Enter your regex: [2-8&&[4-6]]
Enter input string to search: 6
I found the text "6" starting at index 0 and ending at index 1.

Enter your regex: [2-8&&[4-6]]
Enter input string to search: 7
No match found.
```

Subtraction

Finally, you can use *subtraction* to negate one or more nested character classes, such as `[0-9&&[^345]]`. This example creates a single character class that matches everything from 0 to 9, *except* the numbers 3, 4, and 5.

```
Enter your regex: [0-9&&[^345]]
Enter input string to search: 2
I found the text "2" starting at index 0 and ending at index 1.

Enter your regex: [0-9&&[^345]]
Enter input string to search: 3
No match found.

Enter your regex: [0-9&&[^345]]
Enter input string to search: 4
No match found.

Enter your regex: [0-9&&[^345]]
Enter input string to search: 5
No match found.

Enter your regex: [0-9&&[^345]]
Enter input string to search: 6
I found the text "6" starting at index 0 and ending at index 1.

Enter your regex: [0-9&&[^345]]
Enter input string to search: 9
I found the text "9" starting at index 0 and ending at index 1.
```

Predefined Character Classes

The `Pattern` API contains a number of useful *predefined character classes*, which offer convenient shorthands for commonly used regular expressions:

Construct	Description
`.`	Any character (may or may not match line terminators)
`\d`	A digit: `[0-9]`
`\D`	A non-digit: `[^0-9]`
`\s`	A whitespace character: `[ \t\n\x0B\f\r]`
`\S`	A non-whitespace character: `[^\s]`
`\w`	A word character: `[a-zA-Z_0-9]`
`\W`	A non-word character: `[^\w]`

Constructs beginning with a backslash are called *escaped constructs*. We previewed escaped constructs in the String Literals section where we mentioned the use of backslash and `\Q` and `\E` for quotation. If you are using an escaped construct within a string literal, you must precede the backslash with another backslash for the string to compile. For example:

```
private final String REGEX = "\\d"; // a single digit
```

In this example `\d` is the regular expression; the extra backslash is required for the code to compile. The test harness reads the expressions directly from the `Console`, however, so the extra backslash is unnecessary.

The following examples demonstrate the use of predefined character classes.

```
Enter your regex: .
Enter input string to search: @
I found the text "@" starting at index 0 and ending at index 1.

Enter your regex: .
Enter input string to search: 1
I found the text "1" starting at index 0 and ending at index 1.

Enter your regex: .
Enter input string to search: a
I found the text "a" starting at index 0 and ending at index 1.

Enter your regex: \d
Enter input string to search: 1
I found the text "1" starting at index 0 and ending at index 1.

Enter your regex: \d
Enter input string to search: a
No match found.
```

```
Enter your regex: \D
Enter input string to search: 1
No match found.

Enter your regex: \D
Enter input string to search: a
I found the text "a" starting at index 0 and ending at index 1.

Enter your regex: \s
Enter input string to search:
I found the text " " starting at index 0 and ending at index 1.

Enter your regex: \s
Enter input string to search: a
No match found.

Enter your regex: \S
Enter input string to search:
No match found.
```

```
Enter your regex: \S
Enter input string to search: a
I found the text "a" starting at index 0 and ending at index 1.

Enter your regex: \w
Enter input string to search: a
I found the text "a" starting at index 0 and ending at index 1.

Enter your regex: \w
Enter input string to search: !
No match found.

Enter your regex: \W
Enter input string to search: a
No match found.

Enter your regex: \W
Enter input string to search: !
I found the text "!" starting at index 0 and ending at index 1.
```

Quantifiers

Quantifiers allow you to specify the number of occurrences to match against. For convenience, the three sections of the Pattern API specification describing greedy, reluctant, and possessive quantifiers are presented below. At first glance it may appear that the quantifiers `X?`, `X??` and `X?+` do exactly the same thing, since they all promise to match "X, once or not at all". There are subtle implementation differences which will be explained near the end of this section.

Greedy	Reluctant	Possessive	Meaning
`X?`	`X??`	`X?+`	*X*, once or not at all
`X*`	`X*?`	`X*+`	*X*, zero or more times
`X+`	`X+?`	`X++`	*X*, one or more times
`X{n}`	`X{n}?`	`X{n}+`	*X*, exactly *n* times
`X{n,}`	`X{n,}?`	`X{n,}+`	*X*, at least *n* times
`X{n,m}`	`X{n,m}?`	`X{n,m}+`	*X*, at least *n* but not more than *m* times

Differences Among Greedy, Reluctant, and Possessive Quantifiers

There are subtle differences among greedy, reluctant, and possessive quantifiers.

Greedy quantifiers are considered "greedy" because they force the matcher to read in, or *eat*, the entire input string prior to attempting the first match. If the first match attempt (the entire input string) fails, the matcher backs off the input string by one character and tries again, repeating the process until a match is found or there are no more characters left to back off from. Depending on the quantifier used in the expression, the last thing it will try matching against is 1 or 0 characters.

The reluctant quantifiers, however, take the opposite approach: They start at the beginning of the input string, then reluctantly eat one character at a time looking for a match. The last thing they try is the entire input string.

Finally, the possessive quantifiers always eat the entire input string, trying once (and only once) for a match. Unlike the greedy quantifiers, possessive quantifiers never back off, even if doing so would allow the overall match to succeed.

To illustrate, consider the input string `xfooxxxxxxfoo`.

```
Enter your regex: .*foo  // greedy quantifier
Enter input string to search: xfooxxxxxxfoo
I found the text "xfooxxxxxxfoo" starting at index 0 and ending at index 13.

Enter your regex: .*?foo  // reluctant quantifier
Enter input string to search: xfooxxxxxxfoo
I found the text "xfoo" starting at index 0 and ending at index 4.
I found the text "xxxxxxfoo" starting at index 4 and ending at index 13.

Enter your regex: .*+foo // possessive quantifier
Enter input string to search: xfooxxxxxxfoo
No match found.
```

The first example uses the greedy quantifier `.*` to find "anything", zero or more times, followed by the letters `"f" "o" "o"`. Because the quantifier is greedy, the `.*` portion of the expression first eats the entire input string. At this point, the overall expression cannot succeed, because the last three letters (`"f" "o" "o"`) have already been consumed. So the matcher slowly backs off one letter at a time until the rightmost occurrence of "foo" has been regurgitated, at which point the match succeeds and the search ends.

The second example, however, is reluctant, so it starts by first consuming "nothing". Because "foo" doesn't appear at the beginning of the string, it's forced to swallow the first letter (an "x"), which triggers the first match at 0 and 4. Our test harness continues the process until the input string is exhausted. It finds another match at 4 and 13.

The third example fails to find a match because the quantifier is possessive. In this case, the entire input string is consumed by `.*+`, leaving nothing left over to satisfy the "foo" at the end of the expression. Use a possessive quantifier for situations where you want to seize all of something without ever backing off; it will outperform the equivalent greedy quantifier in cases where the match is not immediately found.

Capturing Groups and Character Classes with Quantifiers

Until now, we've only tested quantifiers on input strings containing one character. In fact, quantifiers can only attach to one character at a time, so the regular expression "abc+" would mean "a, followed by b, followed by c one or more times". It would not mean "abc" one or more times. However, quantifiers can also attach to Character Classes and Capturing Groups, such as `[abc]+` (a or b or c, one or more times) or `(abc)+` (the group "abc", one or more times).

Let's illustrate by specifying the group `(dog)`, three times in a row.

```
Enter your regex: (dog){3}
Enter input string to search: dogdogdogdogdogdog
I found the text "dogdogdog" starting at index 0 and ending at index 9.
I found the text "dogdogdog" starting at index 9 and ending at index 18.

Enter your regex: dog{3}
Enter input string to search: dogdogdogdogdogdog
No match found.
```

Similarly, we can apply a quantifier to an entire character class:

```
Enter your regex: [abc]{3}
Enter input string to search: abccabaaaccbbbc
I found the text "abc" starting at index 0 and ending at index 3.
I found the text "cab" starting at index 3 and ending at index 6.
I found the text "aaa" starting at index 6 and ending at index 9.
I found the text "ccb" starting at index 9 and ending at index 12.
I found the text "bbc" starting at index 12 and ending at index 15.

Enter your regex: abc{3}
Enter input string to search: abccabaaaccbbbc
No match found.
```

Here the quantifier {3} applies to the entire character class in the first example, but only to the letter "c" in the second.

Capturing Groups

In the previous section, we saw how quantifiers attach to one character, character class, or capturing group at a time. But until now, we have not discussed the notion of capturing groups in any detail.

Capturing groups are a way to treat multiple characters as a single unit. They are created by placing the characters to be grouped inside a set of parentheses. For example, the regular expression `(dog)` creates a single group containing the letters `"d"` `"o"` and `"g"`. The portion of the input string that matches the capturing group will be saved in memory for later recall via backreferences (as discussed below in the section, Backreferences).

Numbering

As described in the `Pattern` API, 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)`

To find out how many groups are present in the expression, call the `groupCount` method on a matcher object. The `groupCount` method returns an `int` showing the number of capturing groups present in the matcher's pattern. In this example, `groupCount` would return the number 4, showing that the pattern contains 4 capturing groups.

There is also a special group, group 0, which always represents the entire expression. This group is not included in the total reported by `groupCount`. Groups beginning with `(?` are pure, *non-capturing groups* that do not capture text and do not count towards the group total. (You'll see examples of non-capturing groups later in the section Methods of the Pattern Class.)

It's important to understand how groups are numbered because some `Matcher` methods accept an `int` specifying a particular group number as a parameter:

- `public int start(int group)`: Returns the start index of the subsequence captured by the given group during the previous match operation.
- `public int end (int group)`: Returns the index of the last character, plus one, of the subsequence captured by the given group during the previous match operation.
- `public String group (int group)`: Returns the input subsequence captured by the given group during the previous match operation.

Backreferences

The section of the input string matching the capturing group(s) is saved in memory for later recall via *backreference*. A backreference is specified in the regular expression as a backslash (\) followed by a digit indicating the number of the group to be recalled. For example, the expression `(\d\d)` defines one capturing group matching two digits in a row, which can be recalled later in the expression via the backreference `\1`.

To match any 2 digits, followed by the exact same two digits, you would use `(\d\d)\1` as the regular expression:

```
Enter your regex: (\d\d)\1
Enter input string to search: 1212
I found the text "1212" starting at index 0 and ending at index 4.
```

If you change the last two digits the match will fail:

```
Enter your regex: (\d\d)\1
Enter input string to search: 1234
No match found.
```

For nested capturing groups, backreferencing works in exactly the same way: Specify a backslash followed by the number of the group to be recalled.

Boundary Matchers

Until now, we've only been interested in whether or not a match is found *at some location* within a particular input string. We never cared about *where* in the string the match was taking place.

You can make your pattern matches more precise by specifying such information with *boundary matchers*. For example, maybe you're interested in finding a particular word, but only if it appears at the beginning or end of a line. Or maybe you want to know if the match is taking place on a word boundary, or at the end of the previous match.

The following table lists and explains all the boundary matchers.

Boundary Construct	Description
^	The beginning of a line
$	The end of a line
\b	A word boundary
\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

The following examples demonstrate the use of boundary matchers ^ and $. As noted above, ^ matches the beginning of a line, and $ matches the end.

```
Enter your regex: ^dog$
Enter input string to search: dog
I found the text "dog" starting at index 0 and ending at index 3.

Enter your regex: ^dog$
Enter input string to search:       dog
No match found.

Enter your regex: \s*dog$
Enter input string to search:              dog
I found the text "            dog" starting at index 0 and ending at index 15.

Enter your regex: ^dog\w*
Enter input string to search: dogblahblah
I found the text "dogblahblah" starting at index 0 and ending at index 11.
```

To check if a pattern begins and ends on a word boundary (as opposed to a substring within a longer string), just use \b on either side; for example, \bdog\b

```
Enter your regex: \bdog\b
Enter input string to search: The dog plays in the yard.
I found the text "dog" starting at index 4 and ending at index 7.

Enter your regex: \bdog\b
Enter input string to search: The doggie plays in the yard.
No match found.
```

To match the expression on a non-word boundary, use \B instead:

```
Enter your regex: \bdog\B
Enter input string to search: The dog plays in the yard.
No match found.

Enter your regex: \bdog\B
Enter input string to search: The doggie plays in the yard.
I found the text "dog" starting at index 4 and ending at index 7.
```

Creating a Pattern with Flags

The `Pattern` class defines an alternate `compile` method that accepts a set of flags affecting the way the pattern is matched. The flags parameter is a bit mask that may include any of the following public static fields:

- `Pattern.CANON_EQ` Enables canonical equivalence. When this flag is specified, 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. Specifying this flag may impose a performance penalty.
- `Pattern.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.
- `Pattern.COMMENTS` Permits whitespace and comments in the 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)`.
- `Pattern.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.)
- `Pattern.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.
- `Pattern.MULTILINE` Enables multiline mode. In multiline mode the expressions `^` and `$` match just after or just before, respectively, a line terminator or the end of the input sequence. By default these expressions only match at the beginning and the end of the entire input sequence. Multiline mode can also be enabled via the embedded flag expression `(?m)`.
- `Pattern.UNICODE_CASE` Enables Unicode-aware case folding. When this flag is specified then case-insensitive matching, when enabled by the `CASE_INSENSITIVE` flag, is done in a manner consistent with the Unicode Standard. By default, case-insensitive matching assumes that only characters in the US-ASCII charset are being matched. Unicode-aware case folding can also be enabled via the embedded flag expression `(?u)`. Specifying this flag may impose a performance penalty.
- `Pattern.UNIX_LINES` Enables UNIX lines mode. In this mode, only the `'\n'` line terminator is recognized in the behavior of `.`, `^`, and `$`. UNIX lines mode can also be enabled via the embedded flag expression `(?d)`.

In the following steps we will modify the test harness, `RegexTestHarness.java` to create a pattern with case-insensitive matching.

First, modify the code to invoke the alternate version of `compile`:

```
Pattern pattern =
Pattern.compile(console.readLine("%nEnter your regex: "),
Pattern.CASE_INSENSITIVE);
```

Then compile and run the test harness to get the following results:

```
Enter your regex: dog
Enter input string to search: DoGDOg
I found the text "DoG" starting at index 0 and ending at index 3.
I found the text "DOg" starting at index 3 and ending at index 6.
```

As you can see, the string literal "dog" matches both occurences, regardless of case. To compile a pattern with multiple flags, separate the flags to be included using the bitwise OR operator "|". For clarity, the following code samples hardcode the regular expression instead of reading it from the `Console`:

```
pattern = Pattern.compile("[az]$", Pattern.MULTILINE | Pattern.UNIX_LINES);
```

You could also specify an `int` variable instead:

```
final int flags = Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE;
Pattern pattern = Pattern.compile("aa", flags);
```

Embedded Flag Expressions

It's also possible to enable various flags using *embedded flag expressions*. Embedded flag expressions are an alternative to the two-argument version of `compile`, and are specified in the regular expression itself. The following example uses the original test harness, `RegexTestHarness.java` with the embedded flag expression `(?i)` to enable case-insensitive matching.

```
Enter your regex: (?i)foo
Enter input string to search: FOOfooFoOfoO
I found the text "FOO" starting at index 0 and ending at index 3.
I found the text "foo" starting at index 3 and ending at index 6.
I found the text "FoO" starting at index 6 and ending at index 9.
I found the text "foO" starting at index 9 and ending at index 12.
```

Once again, all matches succeed regardless of case.

The embedded flag expressions that correspond to `Pattern`'s publicly accessible fields are presented in the following table:

Constant	Equivalent Embedded Flag Expression
`Pattern.CANON_EQ`	None
`Pattern.CASE_INSENSITIVE`	`(?i)`
`Pattern.COMMENTS`	`(?x)`
`Pattern.MULTILINE`	`(?m)`
`Pattern.DOTALL`	`(?s)`
`Pattern.LITERAL`	None
`Pattern.UNICODE_CASE`	`(?u)`
`Pattern.UNIX_LINES`	`(?d)`

Using the matches(String,CharSequence) Method

The Pattern class defines a convenient matches method that allows you to quickly check if a pattern is present in a given input string. As with all public static methods, you should invoke matches by its class name, such as Pattern.matches("\\d","1");. In this example, the method returns true, because the digit "1" matches the regular expression \d.

Using the split(String) Method

The split method is a great tool for gathering the text that lies on either side of the pattern that's been matched. As shown below in SplitDemo.java, the split method could extract the words "one two three four five" from the string "one:two:three:four:five":

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class SplitDemo {

    private static final String REGEX = ":";
    private static final String INPUT =
        "one:two:three:four:five";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        String[] items = p.split(INPUT);
        for(String s : items) {
            System.out.println(s);
        }
    }
}

OUTPUT:

one
two
three
four
five
```

Other Utility Methods

You may find the following methods to be of some use as well:

- `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 `String s` as if it were a literal pattern. Metacharacters or escape sequences in the input sequence will be given no special meaning.
- `public String toString()` Returns the `String` representation of this pattern. This is the regular expression from which this pattern was compiled.

Methods of the Matcher Class

This section describes some additional useful methods of the `Matcher` class. For convenience, the methods listed below are grouped according to functionality.

Index Methods

Index methods provide useful index values that show precisely where the match was found in the input string:

- `public int start()`: Returns the start index of the previous match.
- `public int start(int group)`: Returns the start index of the subsequence captured by the given group during the previous match operation.
- `public int end()`: Returns the offset after the last character matched.
- `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.

Study Methods

Study methods review the input string and return a boolean indicating whether or not the pattern is found.

- `public boolean lookingAt()`: Attempts to match the input sequence, starting at the beginning of the region, against the pattern.
- `public boolean find()`: Attempts to find the next subsequence of the input sequence that matches the pattern.
- `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.
- `public boolean matches()`: Attempts to match the entire region against the pattern.

Replacement Methods

Replacement methods are useful methods for replacing text in an input string.

- `public Matcher appendReplacement(StringBuffer sb, String replacement)`: Implements a non-terminal append-and-replace step.
- `public StringBuffer appendTail(StringBuffer sb)`: Implements a terminal append-and-replace step.
- `public String replaceAll(String replacement)`: Replaces every subsequence of the input sequence that matches the pattern with the given replacement string.
- `public String replaceFirst(String replacement)`: Replaces the first subsequence of the input sequence that matches the pattern with the given replacement string.
- `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.