Regular Expressions

This lesson explains how to use the java.util.regex API for pattern matching with regular expressions. Although the syntax accepted by this package is similar to the Perl programming language, knowledge of Perl is not a prerequisite.

Package java.util.regex Description

Classes for matching character sequences against patterns specified by regular expressions.
An instance of the Pattern class represents a regular expression that is specified in string form in a syntax similar to that used by Perl.
Instances of the Matcher class are used to match character sequences against a given pattern. Input is provided to matchers via the CharSequence interface in order to support matching against characters from a wide variety of input sources.
Unless otherwise noted, passing a null argument to a method in any class or interface in this package will cause a NullPointerException to be thrown.

Interface Summary

Interface	Description
MatchResult	The result of a match operation.

Class Summary

Class	Description
Matcher	An engine that performs match operations on a **character sequence** by interpreting a **Pattern**.
Pattern	A compiled representation of a regular expression.

Exception Summary

Exception	Description
PatternSyntaxException	Unchecked exception thrown to indicate a syntax error in a regular-expression pattern.

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.

Test Harness

This page defines a reusable test harness, RegexTestHarness.java , for exploring the regular expression constructs supported by this API. The command to run this code is java RegexTestHarness; no command-line arguments are accepted. The application loops repeatedly, prompting the user for a regular expression and input string. Using this test harness is optional, but you may find it convenient for exploring the test cases discussed in the following pages.

```
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

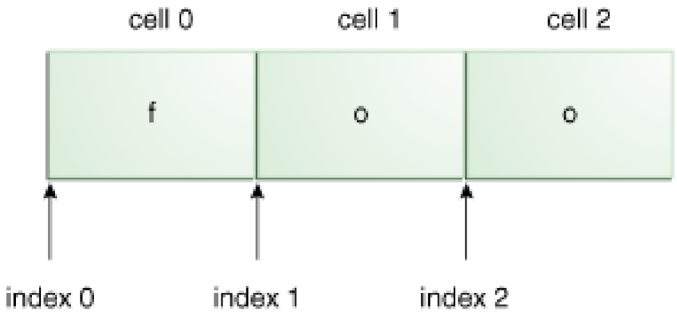
The most basic form of pattern matching supported by this API is the match of a string literal. For example, if the regular expression is foo and the input string is foo, the match will succeed because the strings are identical.

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 figure:



Each character in the string resides in its own cell, with the index positions pointing between each cell. The string "foo" starts at index 0 and ends at index 3, even though the characters themselves only occupy cells 0, 1, and 2.

With subsequent matches, you'll notice some overlap; the start index for the next match is the same as the end index of the previous match:

```
Enter your regex: foo
Enter input string to search: foofoofoo
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.
```

Metacharacters

This API also supports a number of special characters that affect the way a pattern is matched. Change the regular expression to cat. & 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 eg.

Note: In certain situations the special characters listed above will not be treated as metacharacters. You'll encounter this as you learn more about how regular expressions are constructed. You can, however, use this list to check whether or not a specific character will ever be considered a metacharacter. For example, the characters @ and # never carry a special meaning.

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

2 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:

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)

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

Note: The word "class" in the phrase "character class" does not refer to a .class file. In the context of regular expressions, a character class is a set of characters enclosed within square brackets. It specifies the characters that will successfully match a single character from a given input string.

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.
```

In the above examples, the overall match succeeds only when the first letter matches one of the characters defined by the character class.

Character Classes

Negation

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

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.

```
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.
```

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.

Enter your regex: foo[1-5]
Enter input string to search: foo5
I found the text "foo5" starting at index 0 and ending at index 4.

Enter your regex: foo[1-5]
Enter input string to search: foo6
No match found.

Enter your regex: foo[^1-5]
Enter input string to search: foo1
No match found.

Enter your regex: foo[^1-5]
Enter input string to search: foo6
I found the text "foo6" starting at index 0 and ending at index 4.
```

Character Classes

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.
```

Character Classes

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.
```

Now that we've covered how character classes are created,
You may want to review the Character Classes table before
continuing with the next page.

Predefined Character Classes

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

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]
\W	A non-word character: [^\w]

Each construct in the left-hand column is shorthand for the character class in the right-hand column.

Example:
\d means a range of digits (0-9), and \w means a word character (any lowercase letter, any uppercase letter, the underscore character, or any digit). Use the predefined classes whenever possible. They make your code easier to read and eliminate errors introduced by malformed character classes.

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.

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

\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, extra backslash is unnecessary.

In the first 3 examples (on the next page), the regular expression is simply . (the "dot" metacharacter) that indicates "any character." Therefore, the match is successful in all 3 cases (a randomly selected @ character, a digit, and a letter). The remaining examples each use a single regular expression construct from the Predefined Character Classes table.

You can refer to this table to figure out the logic behind each match:

- \d matches all digits
- \s matches spaces
- \w matches word characters

Alternatively, a capital letter means the opposite:

- \D matches non-digits
- \S matches non-spaces
- \W matches non-word characters

Predefined Character Classes

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

Let's start our look at greedy quantifiers by creating three different regular expressions: the letter "a" followed by either ?, *, or +. Let's see what happens when these expressions are tested against an empty input string "":

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

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

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

Quantifiers

Zero-Length Matches

In the above example, the match is successful in the first two cases because the expressions a? and a* both allow for zero occurrences of the letter a. You'll also notice that the start and end indices are both zero, which is unlike any of the examples we've seen so far. The empty input string "" has no length, so the test simply matches nothing at index 0. Matches of this sort are known as a zero-length matches. A zero-length match can occur in several cases: in an empty input string, at the beginning of an input string, after the last character of an input string, or in between any two characters of an input string. Zero-length matches are easily identifiable because they always start and end at the same index position.

Let's explore zero-length matches with a few more examples. Change the input string to a single letter "a" and you'll notice something interesting:

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

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

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

All three quantifiers found the letter "a", but the first two also found a zero-length match at index 1; that is, after the last character of the input string. Remember, the matcher sees the character "a" as sitting in the cell between index 0 and index 1, and our test harness loops until it can no longer find a match. Depending on the quantifier used, the presence of "nothing" at the index after the last character may or may not trigger a match.

Now change the input string to the letter "a" five times in a row and you'll get the following:

```
Enter your regex: a?
Enter input string to search: aaaaa
I found the text "a" starting at index 0 and ending at index 1.
I found the text "a" starting at index 1 and ending at index 2.
I found the text "a" starting at index 2 and ending at index 3.
I found the text "a" starting at index 3 and ending at index 4.
I found the text "a" starting at index 4 and ending at index 5.
I found the text "" starting at index 5 and ending at index 5.

Enter your regex: a*
Enter input string to search: aaaaa
I found the text "aaaaa" starting at index 0 and ending at index 5.
I found the text "" starting at index 5 and ending at index 5.

Enter your regex: a+
Enter input string to search: aaaaa
I found the text "aaaaa" starting at index 0 and ending at index 5.
```

Quantifiers

The expression a? finds an individual match for each character, since it matches when "a" appears zero or one times. The expression a* finds two separate matches: all of the letter "a"'s in the first match, then the zero-length match after the last character at index 5. And finally, a+ matches all occurrences of the letter "a", ignoring the presence of "nothing" at the last index.

At this point, you might be wondering what the results would be if the first two quantifiers encounter a letter other than "a". For example, what happens if it encounters the letter "b", as in "ababaaaab"?

Let's find out:

```
Enter your regex: a?
Enter input string to search: ababaaaab
I found the text "a" starting at index 0 and ending at index 1.
I found the text "" starting at index 1 and ending at index 1.
I found the text "a" starting at index 2 and ending at index 3.
I found the text "" starting at index 3 and ending at index 3.
I found the text "a" starting at index 4 and ending at index 5.
I found the text "a" starting at index 5 and ending at index 6.
I found the text "a" starting at index 6 and ending at index 7.
I found the text "a" starting at index 7 and ending at index 8.
I found the text "" starting at index 8 and ending at index 8.
I found the text "" starting at index 9 and ending at index 9.

Enter your regex: a*
Enter input string to search: ababaaaab
I found the text "a" starting at index 0 and ending at index 1.
I found the text "" starting at index 1 and ending at index 1.
I found the text "a" starting at index 2 and ending at index 3.
I found the text "" starting at index 3 and ending at index 3.
I found the text "aaaa" starting at index 4 and ending at index 8.
I found the text "" starting at index 8 and ending at index 8.
I found the text "" starting at index 9 and ending at index 9.

Enter your regex: a+
Enter input string to search: ababaaaab
I found the text "a" starting at index 0 and ending at index 1.
I found the text "a" starting at index 2 and ending at index 3.
I found the text "aaaa" starting at index 4 and ending at index 8.
```

Even though the letter "b" appears in cells 1, 3, and 8, the output reports a zero-length match at those locations. The regular expression a? is not specifically looking for the letter "b"; it's merely looking for the presence (or lack thereof) of the letter "a". If the quantifier allows for a match of "a" zero times, anything in the input string that's not an "a" will show up as a zero-length match. The remaining a's are matched according to the rules discussed in the previous examples.

To match a pattern exactly n number of times, simply specify the number inside a set of braces:

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

Enter your regex: a{3}
Enter input string to search: aaa
I found the text "aaa" starting at index 0 and ending at index 3.

Enter your regex: a{3}
Enter input string to search: aaaa
I found the text "aaa" starting at index 0 and ending at index 3.
```

Quantifiers

Here, the regular expression a{3} is searching for three occurrences of the letter "a" in a row. The first test fails because the input string does not have enough a's to match against. The second test contains exactly 3 a's in the input string, which triggers a match. The third test also triggers a match because there are exactly 3 a's at the beginning of the input string. Anything following that is irrelevant to the first match. If the pattern should appear again after that point, it would trigger subsequent matches:

```
Enter your regex: a{3}
Enter input string to search: aaaaaaaaa
I found the text "aaa" starting at index 0 and ending at index 3.
I found the text "aaa" starting at index 3 and ending at index 6.
I found the text "aaa" starting at index 6 and ending at index 9.
```

To require a pattern to appear at least n times, add a comma after the number:

```
Enter your regex: a{3,}
Enter input string to search: aaaaaaaaa
I found the text "aaaaaaaaa" starting at index 0 and ending at index 9.
```

With the same input string, this test finds only one match, because the 9 a's in a row satisfy the need for "at least" 3 a's.

Finally, to specify an upper limit on the number of occurrences, add a second number inside the braces:

```
Enter your regex: a{3,6} // find at least 3 (but no more than 6) a's in a row
Enter input string to search: aaaaaaaaa
I found the text "aaaaaa" starting at index 0 and ending at index 6.
I found the text "aaa" starting at index 6 and ending at index 9.
```

Here the first match is forced to stop at the upper limit of 6 characters. The second match includes whatever is left over, which happens to be three a's — the minimum number of characters allowed for this match. If the input string were one character shorter, there would not be a second match since only two a's would remain.

Quantifiers

Capturing Groups & 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. 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.
```

Here the first example finds three matches, since the quantifier applies to the entire capturing group. Remove the parentheses, however, and the match fails because the quantifier {3} now applies only to the letter "g".

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.

Quantifiers

Differences Among Greedy, Reluctant, & 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

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 matchers	
^	The beginning of a line
$	The end of a line
\b	A word boundary
\b{g}	A Unicode extended grapheme cluster 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

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.
```

The first example is successful because the pattern occupies the entire input string. The second example fails because the input string contains extra whitespace at the beginning. The third example specifies an expression that allows for unlimited white space, followed by "dog" on the end of the line. The fourth example requires "dog" to be present at the beginning of a line followed by an unlimited number of word characters.

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.
```

Boundary Matchers

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.
```

To require the match to occur only at the end of the previous match, use \G:

```
Enter your regex: dog
Enter input string to search: dog dog
I found the text "dog" starting at index 0 and ending at index 3.
I found the text "dog" starting at index 4 and ending at index 7.

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

Here the second example finds only one match, because the second occurrence of "dog" does not start at the end of the previous match.

Methods of the Pattern Class

Until now, we've only used the test harness to create Pattern objects in their most basic form. This section explores advanced techniques such as creating patterns with flags and using embedded flag expressions. It also explores some additional useful methods that we haven't yet discussed.

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).

Methods of the Pattern Class

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.

all matches succeed regardless of case.

```
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.
```

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)

Methods of the Pattern Class

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
```

For simplicity, we've matched a string literal, the colon (:) instead of a complex regular expression. Since we're still using Pattern and Matcher objects, you can use split to get the text that falls on either side of any regular expression. Here's the same example, SplitDemo2.java, modified to split on digits instead:

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

public class SplitDemo2 {

    private static final String REGEX = "\\d";
    private static final String INPUT =
        "one9two4three7four1five";

    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
```

Methods of the Pattern Class

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.

Pattern Method Equivalents in java.lang.String

Regular expression support also exists in java.lang.String through several methods that mimic the behavior of java.util.regex.Pattern. For convenience, key excerpts from their API are presented below.

- public boolean matches(String regex): Tells whether or not this string matches the given regular expression. An invocation of this method of the form str.matches(regex) yields exactly the same result as the expression Pattern.matches(regex, str).

- public String[] split(String regex, int limit): Splits this string around matches of the given regular expression. An invocation of this method of the form str.split(regex, n) yields the same result as the expression Pattern.compile(regex).split(str, n)

- public String[] split(String regex): Splits this string around matches of the given regular expression. This method works the same as if you invoked the two-argument split method with the given expression and a limit argument of zero. Trailing empty strings are not included in the resulting array.

There is also a replace method, that replaces one CharSequence with another:

- public String replace(CharSequence target,CharSequence replacement): Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence. The replacement proceeds from the beginning of the string to the end, for example, replacing "aa" with "b" in the string "aaa" will result in "ba" rather than "ab".

Methods of the Matcher Class

methods listed below are grouped according to functionality.

Index

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

Methods:

- 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 Matcherclass. 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.

Methods of the Matcher Class

methods listed below are grouped according to functionality.

Using the start & end Methods

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

public class MatcherDemo {

    private static final String REGEX =
        "\\bdog\\b";
    private static final String INPUT =
        "dog dog dog doggie dogg";

    public static void main(String[] args) {
       Pattern p = Pattern.compile(REGEX);
       //  get a matcher object
       Matcher m = p.matcher(INPUT);
       int count = 0;
       while(m.find()) {
           count++;
           System.out.println("Match number "
                              + count);
           System.out.println("start(): "
                              + m.start());
           System.out.println("end(): "
                              + m.end());
      }
    }
}
```

Here's an example, MatcherDemo.java, that counts the number of times the word "dog" appears in the input string.

```
OUTPUT:

Match number 1
start(): 0
end(): 3
Match number 2
start(): 4
end(): 7
Match number 3
start(): 8
end(): 11
```

You can see that this example uses word boundaries to ensure that the letters "d" "o" "g" are not merely a substring in a longer word. It also gives some useful information about where in the input string the match has occurred. The start method returns the start index of the subsequence captured by the given group during the previous match operation, and end returns the index of the last character matched, plus one.

Using the matches & lookingAt Methods

The matches and lookingAt methods both attempt to match an input sequence against a pattern. The difference, however, is that matches requires the entire input sequence to be matched, while lookingAt does not. Both methods always start at the beginning of the input string. Here's the full code, MatchesLooking.java:

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

public class MatchesLooking {

    private static final String REGEX = "foo";
    private static final String INPUT =
        "fooooooooooooooooo";
    private static Pattern pattern;
    private static Matcher matcher;

    public static void main(String[] args) {

        // Initialize
        pattern = Pattern.compile(REGEX);
        matcher = pattern.matcher(INPUT);

        System.out.println("Current REGEX is: "
                           + REGEX);
        System.out.println("Current INPUT is: "
                           + INPUT);

        System.out.println("lookingAt(): "
            + matcher.lookingAt());
        System.out.println("matches(): "
            + matcher.matches());
    }
}
```

```
Current REGEX is: foo
Current INPUT is: fooooooooooooooooo
lookingAt(): true
matches(): false
```

Methods of the Matcher Class

methods listed below are grouped according to functionality.

Using replaceFirst(String) & replaceAll(String)

The replaceFirst and replaceAll methods replace text that matches a given regular expression. As their names indicate, replaceFirst replaces the first occurrence, and replaceAll replaces all occurrences. Here's the ReplaceDemo.java code:

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

public class ReplaceDemo {

    private static String REGEX = "dog";
    private static String INPUT =
        "The dog says meow. All dogs say meow.";
    private static String REPLACE = "cat";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        // get a matcher object
        Matcher m = p.matcher(INPUT);
        INPUT = m.replaceAll(REPLACE);
        System.out.println(INPUT);
    }
}
```

```
OUTPUT: The cat says meow. All cats say meow.
```

In this first version, all occurrences of dog are replaced with cat. But why stop here? Rather than replace a simple literal like dog, you can replace text that matches any regular expression. The API for this method states that "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-."

Here's the ReplaceDemo2.java code:

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

public class ReplaceDemo2 {

    private static String REGEX = "a*b";
    private static String INPUT =
        "aabfooaabfooabfoob";
    private static String REPLACE = "-";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        // get a matcher object
        Matcher m = p.matcher(INPUT);
        INPUT = m.replaceAll(REPLACE);
        System.out.println(INPUT);
    }
}

OUTPUT: -foo-foo-foo-
```

To replace only the first occurrence of the pattern, simply call replaceFirst instead of replaceAll. It accepts the same parameter.

Methods of the Matcher Class

methods listed below are grouped according to functionality.

Using appendReplacement(StringBuffer,String) & appendTail(StringBuffer)

The Matcher class also provides appendReplacement and appendTail methods for text replacement. The following example, RegexDemo.java, uses these two methods to achieve the same effect as replaceAll.

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

public class RegexDemo {

    private static String REGEX = "a*b";
    private static String INPUT = "aabfooaabfooabfoob";
    private static String REPLACE = "-";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        Matcher m = p.matcher(INPUT); // get a matcher object
        StringBuffer sb = new StringBuffer();
        while(m.find()){
            m.appendReplacement(sb,REPLACE);
        }
        m.appendTail(sb);                                    OUTPUT: -foo-foo-foo-
        System.out.println(sb.toString());
    }
}
```

Matcher Method Equivalents in java.lang.String

For convenience, the String class mimics a couple of Matcher methods as well:

- public String replaceFirst(String regex, String replacement): Replaces the first substring of this string that matches the given regular expression with the given replacement. An invocation of this method of the form str.replaceFirst(regex, repl) yields exactly the same result as the expression Pattern.compile(regex).matcher(str).replaceFirst(repl)

- public String replaceAll(String regex, String replacement): Replaces each substring of this string that matches the given regular expression with the given replacement. An invocation of this method of the form str.replaceAll(regex, repl) yields exactly the same result as the expression Pattern.compile(regex).matcher(str).replaceAll(repl)

Methods of the PatternSyntaxException Class

A PatternSyntaxException is an unchecked exception that indicates a syntax error in a regular expression pattern. The PatternSyntaxException class provides the following methods to help you determine what went wrong:

- `public String getDescription(): Retrieves the description of the error.`
- `public int getIndex(): Retrieves the error index.`
- `public String getPattern(): Retrieves the erroneous regular expression pattern.`
- `public String getMessage(): Returns a multi-line string containing the description of the syntax error and its index, the erroneous regular-expression pattern, and a visual indication of the error index within the pattern.`

The following source code, RegexTestHarness2.java, updates our test harness to check for malformed regular expressions:

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

public class RegexTestHarness2 {

    public static void main(String[] args){
        Pattern pattern = null;
        Matcher matcher = null;

        Console console = System.console();
        if (console == null) {
            System.err.println("No console.");
            System.exit(1);
        }
        while (true) {
            try{
                pattern =
                Pattern.compile(console.readLine("%nEnter your regex: "));

                matcher =
                pattern.matcher(console.readLine("Enter input string to search: "));
            }
            catch(PatternSyntaxException pse){
                console.format("There is a problem" +
                               " with the regular expression!%n");
                console.format("The pattern in question is: %s%n",
                               pse.getPattern());
                console.format("The description is: %s%n",
                               pse.getDescription());
                console.format("The message is: %s%n",
                               pse.getMessage());
                console.format("The index is: %s%n",
                               pse.getIndex());
                System.exit(0);
            }
            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");
            }
        }
    }
}
```

To run this test, enter ?i)foo as the regular expression. This mistake is a common scenario in which the programmer has forgotten the opening parenthesis in the embedded flag expression (?i). Doing so will produce the following results:

```
Enter your regex: ?i)
There is a problem with the regular expression!
The pattern in question is: ?i)
The description is: Dangling meta character '?'
The message is: Dangling meta character '?' near index 0
?i)
^
The index is: 0
```

From this output, we can see that the syntax error is a dangling metacharacter (the question mark) at index 0. A missing opening parenthesis is the culprit.

Unicode Support

As of the JDK 7 release, Regular Expression pattern matching has expanded functionality to support Unicode 6.0.

- Matching a Specific Code Point
- Unicode Character Properties

Matching a Specific Code Point

You can match a specific Unicode code point using an escape sequence of the form \uFFFF, where FFFF is the hexadecimal value of the code point you want to match. For example, \u6771 matches the Han character for east.

Alternatively, you can specify a code point using Perl-style hex notation, \x{...}. For example:

```
String hexPattern = "\x{" + Integer.toHexString(codePoint) + "}";
```

Unicode Character Properties

Each Unicode character, in addition to its value, has certain attributes, or properties. You can match a single character belonging to a particular category with the expression \p{prop}. You can match a single character not belonging to a particular category with the expression \P{prop}.

The three supported property types are scripts, blocks, and a "general" category.

Scripts

To determine if a code point belongs to a specific script, you can either use the script keyword, or the sc short form, for example, \p{script=Hiragana}. Alternatively, you can prefix the script name with the string Is, such as \p{IsHiragana}.

Valid script names supported by Pattern are those accepted by UnicodeScript.forName.

Blocks

A block can be specified using the block keyword, or the blk short form, for example, \p{block=Mongolian}. Alternatively, you can prefix the block name with the string In, such as \p{InMongolian}.
Valid block names supported by Pattern are those accepted by UnicodeBlock.forName.

General Category

Categories can be specified with optional prefix Is. For example, IsL matches the category of Unicode letters. Categories can also be specified by using the general_category keyword, or the short form gc. For example, an uppercase letter can be matched using general_category=Lu or gc=Lu.
Supported categories are those of The Unicode Standard in the version specified by the Character class.