**Regular Expressions Syntax**

A regular expression, or regex, is a sequence of characters that specifies a pattern which can be searched for in a text. A regex defines a set of strings, usually united for a given purpose. Suppose you need a way to formalize and refer to all the strings that make up the format of an email address. Since there are a near infinite number of possible email addresses, it’d be hard to enumerate them all. However, as we know an email address has a specific structure, and we can encode that using the regex syntax.

**Java Regex syntax**

Let’s move on now to the syntax for Java regular expressions. The Pattern.compile method takes a String, which is the regex that defines a set of matching strings. Naturally, it has to have a tricky syntax, otherwise a single string defining the pattern can only represent itself.

A regular character in the regex syntax matches that character in the text. If you’ll create a Pattern with Pattern.compile("a") it will only match only the String “a”. There is also an escape character, which is the backslash “\”. It is used to distinguish when the pattern contains an instruction in the syntax or a character. Let’s look at an example as to why we need an escape character. Imagine “[” has a special meaning in the regular expression syntax (it has). How can you determine if “[” is a command to the matching engine or a pattern containing only the bracket? You cannot, so to specify the characters that are also the commands in the syntax you need to escape them. It means “\[” is a pattern for the string “[“, and “[” is part of a command. What about trying to match a backslash? You need to escape it too, so be prepared to see something like “\\\\” in the regex code.

**Character classes**

On top of specifying the expressions that contain individual characters only, you can define the whole classes of characters. Think of them as sets, if a character in some text belongs to the character class, it is matched.

Here is a table with the most used character classes.

* [abc] simple, matches a or b, or c.
* [\^abc] negation, matches everything except a, b, or c.
* [a-c] range, matches a or b, or c.
* [a-c[f-h]] union, matches a, b, c, f, g, h.

For your convenience, there are some useful classes defined already. For example, digits are a perfect example of a useful character class. For example a 5 digit number could be coded into a pattern as “[0-9][0-9][0-9][0-9][0-9]”, but it’s quite ugly. So there’s a shorthand for that: “\d”.

Here are the other classes you need to know.

* . Any character
* \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]

Note that the letter specifying a predefined character class is typically lowercase, the uppercase version tends to mean the negation of the class.

**Boundary matchers**

Next, there’s syntax to specify the position of the matched sequence in the original text you’re searching. If you only need to filter out the strings that start with an email address or something, this is extremely useful.

* ^ 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.

A noteworthy combination of the boundary matchers is the “^pattern$” which will only match the text if it is the full pattern.

**Logical operations**

Now we’re getting into more advanced territory. If a pattern is more than a single character long, it will match a longer string too. In general “XY” in the regex syntax matches X followed by Y. However, there’s also an OR operation, denoted by the post “|”. The “X|Y” regex means it is either X or Y. This is a very powerful feature; you can combine the character classes or sequences of characters (include them in brackets).

**Quantifiers**

On top of everything, you can say how many times the sequence of characters can be repeated for the match. The regex “1” only matches the input “1”, but if we need to match a string of any length consisting of the character “1” you need to use one of the following quantifiers.

* \* matches zero or more occurrences.
* + matches one or more occurrences.
* ? matches zero or one occurrence.

**Groups and backreferences**

A group is a captured subsequence of characters which may be used later in the expression with a backreference. We’ve mentioned already that if you enclose a group of characters in parentheses, you can apply quantifiers or logical or to the whole group. What is even more awesome is that you can refer to the actual characters in the text matched by the group, later.  
Here’s how you do it:  
(...) – defines a group, in the parantheses  
\N – refers to a matched group  
For example:  
(\d\d) – a group of two digits  
(\d\d)/\1– two digits repeated twice, \1 – refers to the matched group, so this regular expression matches the strings that look like “aabb” where a and b are both digits.

**1.1. Character Classes**

Character classes are used to define the content of the pattern. E.g. what should the pattern look for?

|  |
| --- |
| . Dot, any character (may or may not match line terminators, read on)  \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] |

However; notice that in Java, you will need to “double escape” these backslashes.

|  |
| --- |
| String pattern = "**\\**d **\\**D **\\**W **\\**w **\\**S **\\**s"; |

**Quantifiers**

Quantifiers can be used to specify the number or length that part of a pattern should match or repeat. A quantifier will bind to the expression group to its immediate left.

|  |
| --- |
| \* Match 0 or more times  + Match 1 or more times  ? Match 1 or 0 times  {n} Match exactly n times  {n,} Match at least n times  {n,m} Match at least n but not more than m times |

**1.3. Meta-characters**

Meta-characters are used to group, divide, and perform special operations in patterns.

|  |
| --- |
| \ Escape the next meta-character (it becomes a normal/literal character)  ^ Match the beginning of the line  . Match any character (except newline)  $ Match the end of the line (or before newline at the end)  | Alternation (‘or’ statement)  () Grouping  [] Custom character class |

**Pattern flags**

Remember when we talked about the useful API in Java for the regular expressions, there was a method to compile a pattern that took the flags. These will control how the pattern behaves.

Here are some flags that can be useful here and there.

* Pattern.CASE\_INSENSITIVE – enables case-insensitive matching
* Pattern.COMMENTS – whitespace and comments starting with # are ignored until the end of a line.
* Pattern.MULTILINE – one expression can match multiple lines.
* Pattern.DOTALL – the expression “.” matches any character, including a line terminator
* Pattern.UNIX\_LINES – only the ‘\n’ line terminator is recognized in the behavior of ., ^, and $.

## Examples

#### 2.1. Basic Expressions

Every string is a regular expression. For example, the string, “I lost my wallet”, is a regular expression that will match the text, “I lost my wallet”, and will ignore everything else. What if we want to be able to find more things that we lost? We can replace wallet with a character class expression that will match any word.

|  |
| --- |
| "I lost my **\\**w+" |

As you can see, this pattern uses both a **character class** and a **quantifier**. “\w” says match a word character, and “+” says match one or more. So when combined, the pattern says “match one or more word characters.” Now the pattern will match any word in place of “wallet”. E.g. “I lost my sablefish”, “I lost my parrot”, but it will not match “I lost my: trooper”, because as soon as the expression finds the ":" character, which is not a word character, it will stop matching. If we want the expression to be able to handle this situation, then we need to make a small change.

|  |
| --- |
| "I lost my:? **\\**w+" |

Now the expression will allow an optional ":" directly after the word ‘my’.

#### 

#### 2.2. Basic Grouping

An important feature of regular expressions is the ability to group sections of a pattern, and provide alternate matches.

|  |
| --- |
| | Alternation (‘or’ statement)  () Grouping |

These two meta-characters are core parts of flexible regular expressions. For instance, in the first example we lost our wallet. What if we knew exactly which types of objects we had lost, and we wanted to find those objects but nothing else? We can use a group (), with an ‘or’ meta-character in order to specify a list of expressions to allow in our match.

|  |
| --- |
| "I lost my:? (wallet|car|cell phone|marbles)" |

The new expression will now match the beginning of the string “I lost my”, an optional ":", and then any one of the expressions in the group, separated by alternators, "|"; any one of the following: ‘wallet’, ‘cell phone’, ‘car’, or our ‘marbles’ would be a match.

|  |
| --- |
| "I lost my wallet" matches  "I lost my wallets" matches the ‘s’ is not needed, is ignored  "I lost my: car" matches  "I lost my- car" doesn’t match ‘-‘ is not allowed in our pattern  "I lost my: cell" doesn’t match all of ‘cell phone’ is needed  "I lost my: cell phone" matches  "I lost my cell phone" matches  "I lost my marbles" matches |

As you can see, the combinations for matches quickly become very large. This is not the complete set, as there are several more phrases that would match our simple pattern.

Can you figure out all possible matches for this pattern?

|  |
| --- |
| "I lost my:? (wallet|car|cell phone|marbles)" |

**Answer:** This is a trick question! Because this regular expression is unlimited (has no beginning `^` and no ending `$` meta-characters to terminate the match,) the pattern we’ve created will actually match any string containing one of the results below. In short, nearly infinite possible matches; however, if we did want to limit our pattern to just these results, we could use add the required terminators to our pattern – like so:

|  |  |
| --- | --- |
| "^I lost my:? (wallet|car|cell phone|marbles)$" | |
| "I lost my wallet"  "I lost my wallets"  "I lost my: wallet"  "I lost my: wallets"  "I lost my car"  "I lost my car"  "I lost my: car"  "I lost my: car"  "I lost my cell phone"  "I lost my cell phone"  "I lost my: cell phone"  "I lost my: cell phone"  "I lost my marbles"  "I lost my marbles"  "I lost my: marbles"  "I lost my: marbles" |

#### 2.3. Matching/Validating

Regular expressions make it possible to find all instances of text that match a certain pattern, and return a Boolean value if the pattern is found/not found. (This can be used to validate input such as phone numbers, social security numbers, email addresses, web form input data, scrub data, and much more. Eg. If the pattern is found in a String, and the pattern matches a SSN, then the string is an SSN)

Sample code

import java.util.ArrayList;

import java.util.List;

public class ValidateDemo {

public static void main(String[] args) {

List<String> input = new ArrayList<String>();

input.add("123-45-6789");

input.add("9876-5-4321");

input.add("987-65-4321 (attack)");

input.add("987-65-4321 ");

input.add("192-83-7465");

for (String ssn : input) {

if (ssn.matches("^(\\d{3}-?\\d{2}-?\\d{4})$")) {

System.out.println("Found good SSN: " + ssn);

}

}

}

}

**This produces the following output:**

|  |
| --- |
| Found good SSN: 123-45-6789</br>  Found good SSN: 192-83-7465 |

##### Dissecting the pattern:

|  |
| --- |
| "^(**\\**d{3}-?**\\**d{2}-?**\\**d{4})$" |
| ^ match the beginning of the line  () group everything within the parenthesis as group 1  \d{n} match n digits, where n is a number equal to or greater than zero  -? optionally match a dash  $ match the end of the line | |

The java.util.regex package primarily consists of the following three classes −

* **Pattern Class** − 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.
* **Matcher Class** − 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.
* **PatternSyntaxException** − A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.

## 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".

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

* ((A)(B(C)))
* (A)
* (B(C))
* (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.

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.

**Example**

Following example illustrates how to find a digit string from the given alphanumeric string −

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches {

public static void main( String args[] ) {

// String to be scanned to find the pattern.

String line = "This order was placed for QT3000! OK?";

String pattern = "(.\*)(\\d+)(.\*)";

// Create a Pattern object

Pattern r = Pattern.compile(pattern);

// Now create matcher object.

Matcher m = r.matcher(line);

if (m.find( )) {

System.out.println("Found value: " + m.group(0) );

System.out.println("Found value: " + m.group(1) );

System.out.println("Found value: " + m.group(2) );

}else {

System.out.println("NO MATCH");

}

}

}

This will produce the following result −

**Output**

Found value: This order was placed for QT3000! OK?

Found value: This order was placed for QT300

Found value: 0

## Regular Expression Syntax

Here is the table listing down all the regular expression metacharacter syntax available in Java −

|  |  |
| --- | --- |
| **Subexpression** | **Matches** |
| ^ | Matches the beginning of the line. |
| $ | Matches the end of the line. |
| . | Matches any single character except newline. Using **m** option allows it to match the newline as well. |
| [...] | Matches any single character in brackets. |
| [^...] | Matches any single character not in brackets. |
| \A | Beginning of the entire string. |
| \z | End of the entire string. |
| \Z | End of the entire string except allowable final line terminator. |
| re\* | Matches 0 or more occurrences of the preceding expression. |
| re+ | Matches 1 or more of the previous thing. |
| re? | Matches 0 or 1 occurrence of the preceding expression. |
| re{ n} | Matches exactly n number of occurrences of the preceding expression. |
| re{ n,} | Matches n or more occurrences of the preceding expression. |
| re{ n, m} | Matches at least n and at most m occurrences of the preceding expression. |
| a| b | Matches either a or b. |
| (re) | Groups regular expressions and remembers the matched text. |
| (?: re) | Groups regular expressions without remembering the matched text. |
| (?> re) | Matches the independent pattern without backtracking. |
| \w | Matches the word characters. |
| \W | Matches the nonword characters. |
| \s | Matches the whitespace. Equivalent to [\t\n\r\f]. |
| \S | Matches the nonwhitespace. |
| \d | Matches the digits. Equivalent to [0-9]. |
| \D | Matches the nondigits. |
| \A | Matches the beginning of the string. |
| \Z | Matches the end of the string. If a newline exists, it matches just before newline. |
| \z | Matches the end of the string. |
| \G | Matches the point where the last match finished. |
| \n | Back-reference to capture group number "n". |
| \b | Matches the word boundaries when outside the brackets. Matches the backspace (0x08) when inside the brackets. |
| \B | Matches the nonword boundaries. |
| \n, \t, etc. | Matches newlines, carriage returns, tabs, etc. |
| \Q | Escape (quote) all characters up to \E. |
| \E | Ends quoting begun with \Q. |

## Methods of the Matcher Class

Here is a list of useful instance methods −

### Index Methods

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

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public int start()**  Returns the start index of the previous match. |
| 2 | **public int start(int group)**  Returns the start index of the subsequence captured by the given group during the previous match operation. |
| 3 | **public int end()**  Returns the offset after the last character matched. |
| 4 | **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 −

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public boolean lookingAt()**  Attempts to match the input sequence, starting at the beginning of the region, against the pattern. |
| 2 | **public boolean find()**  Attempts to find the next subsequence of the input sequence that matches the pattern. |
| 3 | **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. |
| 4 | **public boolean matches()**  Attempts to match the entire region against the pattern. |

### The start and end Methods

Following is the example that counts the number of times the word "cat" appears in the input string −

**Example**

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches {

private static final String REGEX = "\\bcat\\b";

private static final String INPUT = "cat cat cat cattie cat";

public static void main( String args[] ) {

Pattern p = Pattern.compile(REGEX);

Matcher m = p.matcher(INPUT); // get a matcher object

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());

}

}

}

This will produce the following result −

**Output**

Match number 1

start(): 0

end(): 3

Match number 2

start(): 4

end(): 7

Match number 3

start(): 8

end(): 11

Match number 4

start(): 19

end(): 22

You can see that this example uses word boundaries to ensure that the letters "c" "a" "t" 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 the end returns the index of the last character matched, plus one.

### The matches and 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 is the example explaining the functionality −

**Example**

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches {

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[] ) {

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());

}

}

This will produce the following result −

**Output**

Current REGEX is: foo

Current INPUT is: fooooooooooooooooo

lookingAt(): true

matches(): false

## java.util.regex.Pattern class:

#### 1) Pattern.matches()

We have already seen the usage of this method in the above example where we performed the search for string “book” in a given text. This is one of simplest and easiest way of searching a String in a text using Regex.

String content = "This is a tutorial Website!";

String patternString = ".\*tutorial.\*";

boolean isMatch = Pattern.matches(patternString, content);

System.out.println("The text contains 'tutorial'? " + isMatch);

As you can see we have used matches() method of Pattern class to search the pattern in the given text. The pattern .\*tutorial.\* allows zero or more characters at the beginning and end of the String “tutorial” (the expression .\* is used for zero and more characters).

**Limitations**: This way we can search a single occurrence of a pattern in a text. For matching multiple occurrences you should use the Pattern.compile() method (discussed in the next section).

#### 2) Pattern.compile()

In the above example we searched a string “tutorial” in the text, that is a case sensitive search, however if you want to do a CASE INSENSITIVE search or want to do search multiple occurrences then you may need to first compile the pattern using Pattern.compile() before searching it in text. This is how this method can be used for this case.

String content = "This is a tutorial Website!";

String patternString = ".\*tuToRiAl.";

Pattern pattern = Pattern.compile(patternString, Pattern.CASE\_INSENSITIVE);

Here we have used a flag Pattern.CASE\_INSENSITIVE for case insensitive search, there are several other flags that can be used for different-2 purposes. To read more about such flags [refer this document](https://docs.oracle.com/javase/tutorial/essential/regex/pattern.html).

**Now what**: We have obtained a Pattern instance but how to match it? For that we would be needing a Matcher instance, which we can get using Pattern.matcher() method. Lets discuss it.

#### 3) Pattern.matcher() method

In the above section we learnt how to get a Pattern instance using compile() method. Here we will learn How to get Matcher instance from Pattern instance by using matcher() method.

String content = "This is a tutorial Website!";

String patternString = ".\*tuToRiAl.\*";

Pattern pattern = Pattern.compile(patternString, Pattern.CASE\_INSENSITIVE);

Matcher matcher = pattern.matcher(content);

boolean isMatched = matcher.matches();

System.out.println("Is it a Match?" + isMatched);

Output:

Is it a Match?true

#### 4) Pattern.split()

To split a text into multiple strings based on a delimiter (Here delimiter would be specified using **regex**), we can use Pattern.split() method. This is how it can be done.

import java.util.regex.\*;

class RegexExample2{

public static void main(String args[]){

String text = "ThisIsChaitanya.ItISMyWebsite";

// Pattern for delimiter

String patternString = "is";

Pattern pattern = Pattern.compile(patternString, Pattern.CASE\_INSENSITIVE);

String[] myStrings = pattern.split(text);

for(String temp: myStrings){

System.out.println(temp);

}

System.out.println("Number of split strings: "+myStrings.length);

}}

Output:

Th

Chaitanya.It

MyWebsite

Number of split strings: 4

The second split String is null in the output.

## java.util.regex.Matcher Class

We already discussed little bit about Matcher class above. Lets recall few things:

#### Creating a Matcher instance

String content = "Some text";

String patternString = ".\*somestring.\*";

Pattern pattern = Pattern.compile(patternString);

Matcher matcher = pattern.matcher(content);

#### Main methods

**matches()**: It matches the regular expression against the whole text passed to the Pattern.matcher() method while creating Matcher instance.

...

Matcher matcher = pattern.matcher(content);

boolean isMatch = matcher.matches();

**lookingAt()**: Similar to matches() method except that it matches the regular expression only against the beginning of the text, while matches() search in the whole text.

**find()**: Searches the occurrences of of the regular expressions in the text. Mainly used when we are searching for multiple occurrences.

**start() and end()**: Both these methods are generally used along with the find() method. They are used for getting the start and end indexes of a match that is being found using find() method.

#### Lets take an example to find out the multiple occurrences using Matcher methods:

package beginnersbook.com;

import java.util.regex.\*;

class RegexExampleMatcher{

public static void main(String args[]){

String content = "ZZZ AA PP AA QQQ AAA ZZ";

String string = "AA";

Pattern pattern = Pattern.compile(string);

Matcher matcher = pattern.matcher(content);

while(matcher.find()) {

System.out.println("Found at: "+ matcher.start()

+

" - " + matcher.end());

}

}

}

Output:

Found at: 4 - 6

Found at: 10 - 12

Found at: 17 - 19

Now we are familiar with Pattern and Matcher class and the process of matching a regular expression against the text. Lets see what kind of various options we have to define a regular expression:

#### 1) String Literals

Lets say you just want to search a particular string in the text for e.g. “abc” then we can simply write the code like this: Here text and regex both are same.  
Pattern.matches("abc", "abc")

#### 2) Character Classes

A character class matches a single character in the input text against multiple allowed characters in the character class. For example [Cc]haitanya would match all the occurrences of String “chaitanya” with either lower case or upper case C”. Few more examples:  
Pattern.matches("[pqr]", "abcd"); It would give false as no p,q or r in the text  
Pattern.matches("[pqr]", "r"); Return true as r is found  
Pattern.matches("[pqr]", "pq"); Return false as any one of them can be in text not both.

Here is the complete list of various character classes constructs:  
[abc]: It would match with text if the text is having either one of them(a,b or c) and only once.  
[^abc]:  Any single character except a, b, or c (^ denote 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]]:  Any one of them (d, e, or f)  
[a-z&&[^bc]]: a through z, except for b and c: [ad-z] (subtraction)  
[a-z&&[^m-p]]:  a through z, and not m through p: [a-lq-z] (subtraction)

#### Predefined Character Classes – Metacharacters

These are like short codes which you can use while writing regex.

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]

For e.g.  
Pattern.matches("\\d", "1"); would return true  
Pattern.matches("\\D", "z"); return true  
Pattern.matches(".p", "qp"); return true, dot(.) represent any character

#### Boundary Matchers

^ Matches the beginning of a line.

$ Matches then end of a line.

\b Matches a word boundary.

\B Matches a non-word boundary.

\A Matches the beginning of the input text.

\G Matches the end of the previous match

\Z Matches the end of the input text except the final terminator if any.

\z Matches the end of the input text.

For e.g.  
Pattern.matches("^Hello$", "Hello"): return true, Begins and ends with Hello  
Pattern.matches("^Hello$", "Namaste! Hello"): return false, does not begin with Hello  
Pattern.matches("^Hello$", "Hello Namaste!"): return false, Does not end with Hello

#### Quantifiers

Greedy Reluctant Possessive Matches

X? X?? X?+ Matches X once, or not at all (0 or 1 time).

X\* X\*? X\*+ Matches X zero or more times.

X+ X+? X++ Matches X one or more times.

X{n} X{n}? X{n}+ Matches X exactly n times.

X{n,} X{n,}? X{n,}+ Matches X at least n times.

X{n, m) X{n, m)? X{n, m)+ Matches X at least n time, but at most m times.

#### Few examples

import java.util.regex.\*;

class RegexExample{

public static void main(String args[]){

// It would return true if string matches exactly "tom"

System.out.println(

Pattern.matches("tom", "Tom")); //False

/\* returns true if the string matches exactly

\* "tom" or "Tom"

\*/

System.out.println(

Pattern.matches("[Tt]om", "Tom")); //True

System.out.println(

Pattern.matches("[Tt]om", "Tom")); //True

/\* Returns true if the string matches exactly "tim"

\* or "Tim" or "jin" or "Jin"

\*/

System.out.println(

Pattern.matches("[tT]im|[jJ]in", "Tim"));//True

System.out.println(

Pattern.matches("[tT]im|[jJ]in", "jin"));//True

/\* returns true if the string contains "abc" at

\* any place

\*/

System.out.println(

Pattern.matches(".\*abc.\*", "deabcpq"));//True

/\* returns true if the string does not have a

\* number at the beginning

\*/

System.out.println(

Pattern.matches("^[^\\d].\*", "123abc")); //False

System.out.println(

Pattern.matches("^[^\\d].\*", "abc123")); //True

// returns true if the string contains of three letters

System.out.println(

Pattern.matches("[a-zA-Z][a-zA-Z][a-zA-Z]", "aPz"));//True

System.out.println(

Pattern.matches("[a-zA-Z][a-zA-Z][a-zA-Z]", "aAA"));//True

System.out.println(

Pattern.matches("[a-zA-Z][a-zA-Z][a-zA-Z]", "apZx"));//False

// returns true if the string contains 0 or more non-digits

System.out.println(

Pattern.matches("\\D\*", "abcde")); //True

System.out.println(

Pattern.matches("\\D\*", "abcde123")); //False

/\* Boundary Matchers example

\* ^ denotes start of the line

\* $ denotes end of the line

\*/

System.out.println(

Pattern.matches("^This$", "This is Chaitanya")); //False

System.out.println(

Pattern.matches("^This$", "This")); //True

System.out.println(

Pattern.matches("^This$", "Is This Chaitanya")); //False

}

}

A simple example for a regular expression is a (literal) string. For example, the Hello World regex matches the "Hello World" string. . (dot) is another example for a regular expression. A dot matches any single character; it would match, for example, "a" or "1".

The following tables lists several regular expressions and describes which pattern they would match.

| Table 1. Regex example | |
| --- | --- |
| **Regex** | **Matches** |
| this is text | Matches exactly "this is text" |
| this\s+is\s+text | Matches the word "this" followed by one or more whitespace characters followed by the word "is" followed by one or more whitespace characters followed by the word "text". |
| ^\d+(\.\d+)? | ^ defines that the patter must start at beginning of a new line. \d+ matches one or several digits. The ? makes the statement in brackets optional. \. matches ".", parentheses are used for grouping. Matches for example "5", "1.5" and "2.21". |

## [3. Rules of writing regular expressions](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#rules-of-writing-regular-expressions)

The following description is an overview of available meta characters which can be used in regular expressions. This chapter is supposed to be a references for the different regex elements.

### [3.1. Common matching symbols](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#common-matching-symbols)

| **Regular Expression** | **Description** |
| --- | --- |
| . | Matches any character |
| ^regex | Finds regex that must match at the beginning of the line. |
| regex$ | Finds regex that must match at the end of the line. |
| [abc] | Set definition, can match the letter a or b or c. |
| [abc][vz] | Set definition, can match a or b or c followed by either v or z. |
| [^abc] | When a caret appears as the first character inside square brackets, it negates the pattern. This pattern matches any character except a or b or c. |
| [a-d1-7] | Ranges: matches a letter between a and d and figures from 1 to 7, but not d1. |
| X|Z | Finds X or Z. |
| XZ | Finds X directly followed by Z. |
| $ | Checks if a line end follows. |

### [3.2. Meta characters](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#meta-characters)

The following meta characters have a pre-defined meaning and make certain common patterns easier to use, e.g., \d instead of [0..9].

| **Regular Expression** | **Description** |
| --- | --- |
| \d | Any digit, short for [0-9] |
| \D | A non-digit, short for [^0-9] |
| \s | A whitespace character, short for [ \t\n\x0b\r\f] |
| \S | A non-whitespace character, short for |
| \w | A word character, short for [a-zA-Z\_0-9] |
| \W | A non-word character [^\w] |
| \S+ | Several non-whitespace characters |
| \b | Matches a word boundary where a word character is [a-zA-Z0-9\_] |

|  |  |
| --- | --- |
|  | These meta characters have the same first letter as their representation, e.g., digit, space, word, and boundary. Uppercase symbols define the opposite. |

### [3.3. Quantifier](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#quantifier)

A quantifier defines how often an element can occur. The symbols ?, \*, + and {} define the quantity of the regular expressions

| **Regular Expression** | **Description** | **Examples** |
| --- | --- | --- |
| \* | Occurs zero or more times, is short for {0,} | X\* finds no or several letter X, <sbr /> .\* finds any character sequence |
| + | Occurs one or more times, is short for {1,} | X+- Finds one or several letter X |
| ? | Occurs no or one times, ? is short for {0,1}. | X? finds no or exactly one letter X |
| {X} | Occurs X number of times, {} describes the order of the preceding liberal | \d{3} searches for three digits, .{10} for any character sequence of length 10. |
| {X,Y} | Occurs between X and Y times, | \d{1,4} means \d must occur at least once and at a maximum of four. |
| \*? | ? after a quantifier makes it a reluctant quantifier. It tries to find the smallest match. This makes the regular expression stop at the first match. |  |

### [3.4. Grouping and back reference](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#grouping-and-back-reference)

You can group parts of your regular expression. In your pattern you group elements with round brackets, e.g., (). This allows you to assign a repetition operator to a complete group.

In addition these groups also create a back reference to the part of the regular expression. This captures the group. A back reference stores the part of the String which matched the group. This allows you to use this part in the replacement.

Via the $ you can refer to a group. $1 is the first group, $2 the second, etc.

Let’s, for example, assume you want to replace all whitespace between a letter followed by a point or a comma. This would involve that the point or the comma is part of the pattern. Still it should be included in the result.

// Removes whitespace between a word character and . or ,

String pattern = "(\\w)(\\s+)([\\.,])";

System.out.println(EXAMPLE\_TEST.replaceAll(pattern, "$1$3"));

This example extracts the text between a title tag.

// Extract the text between the two title elements

pattern = "(?i)(<title.\*?>)(.+?)()";

String updated = EXAMPLE\_TEST.replaceAll(pattern, "$2");

### [3.5. Negative look ahead](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#negative-look-ahead)

Negative look ahead provides the possibility to exclude a pattern. With this you can say that a string should not be followed by another string.

Negative look ahead are defined via (?!pattern). For example, the following will match "a" if "a" is not followed by "b".

a(?!b)

### [3.6. Specifying modes inside the regular expression](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#specifying-modes-inside-the-regular-expression)

You can add the mode modifiers to the start of the regex. To specify multiple modes, simply put them together as in (?ismx).

* (?i) makes the regex case insensitive.
* (?s) for "single line mode" makes the dot match all characters, including line breaks.
* (?m) for "multi-line mode" makes the caret and dollar match at the start and end of each line in the subject string.

### [3.7. Backslashes in Java](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#backslashes-in-java)

The backslash \ is an escape character in Java Strings. That means backslash has a predefined meaning in Java. You have to use double backslash \\ to define a single backslash. If you want to define \w, then you must be using \\w in your regex. If you want to use backslash as a literal, you have to type \\\\ as \ is also an escape character in regular expressions.

## [4. Using regular expressions with String methods](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#using-regular-expressions-with-string-methods)

### [4.1. Redefined methods on String for processing regular expressions](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#redefined-methods-on-string-for-processing-regular-expressions)

Strings in Java have built-in support for regular expressions. Strings have four built-in methods for regular expressions, i.e., the matches(), split()), replaceFirst() and replaceAll() methods. The replace() method does NOT support regular expressions.

These methods are not optimized for performance. We will later use classes which are optimized for performance.

| **Method** | **Description** |
| --- | --- |
| s.matches("regex") | Evaluates if "regex" matches s. Returns only true if the WHOLE string can be matched. |
| s.split("regex") | Creates an array with substrings of s divided at occurrence of "regex". "regex" is not included in the result. |
| s.replaceFirst("regex"), "replacement" | Replaces first occurance of "regex" with "replacement. |
| s.replaceAll("regex"), "replacement" | Replaces all occurances of "regex" with "replacement. |

Create for the following example the Java project de.vogella.regex.test.

package de.vogella.regex.test;

public class RegexTestStrings {

public static final String EXAMPLE\_TEST = "This is my small example "

+ "string which I'm going to " + "use for pattern matching.";

public static void main(String[] args) {

System.out.println(EXAMPLE\_TEST.matches("\\w.\*"));

String[] splitString = (EXAMPLE\_TEST.split("\\s+"));

System.out.println(splitString.length);// should be 14

for (String string : splitString) {

System.out.println(string);

}

// replace all whitespace with tabs

System.out.println(EXAMPLE\_TEST.replaceAll("\\s+", "\t"));

}

}

### [4.2. Examples](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#examples)

The following class gives several examples for the usage of

regular expressions with strings. See the comment for the purpose.

If you want to test these examples, create for the Java project de.vogella.regex.string.

package de.vogella.regex.string;

public class StringMatcher {

// returns true if the string matches exactly "true"

public boolean isTrue(String s){

return s.matches("true");

}

// returns true if the string matches exactly "true" or "True"

public boolean isTrueVersion2(String s){

return s.matches("[tT]rue");

}

// returns true if the string matches exactly "true" or "True"

// or "yes" or "Yes"

public boolean isTrueOrYes(String s){

return s.matches("[tT]rue|[yY]es");

}

// returns true if the string contains exactly "true"

public boolean containsTrue(String s){

return s.matches(".\*true.\*");

}

// returns true if the string contains of three letters

public boolean isThreeLetters(String s){

return s.matches("[a-zA-Z]{3}");

// simpler from for

// return s.matches("[a-Z][a-Z][a-Z]");

}

// returns true if the string does not have a number at the beginning

public boolean isNoNumberAtBeginning(String s){

return s.matches("^[^\\d].\*");

}

// returns true if the string contains a arbitrary number of characters except b

public boolean isIntersection(String s){

return s.matches("([\\w&&[^b]])\*");

}

// returns true if the string contains a number less than 300

public boolean isLessThenThreeHundred(String s){

return s.matches("[^0-9]\*[12]?[0-9]{1,2}[^0-9]\*");

}

}

And a small JUnit Test to validates the examples.

package de.vogella.regex.string;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.assertFalse;

import static org.junit.Assert.assertTrue;

public class StringMatcherTest {

private StringMatcher m;

@Before

public void setup(){

m = new StringMatcher();

}

@Test

public void testIsTrue() {

assertTrue(m.isTrue("true"));

assertFalse(m.isTrue("true2"));

assertFalse(m.isTrue("True"));

}

@Test

public void testIsTrueVersion2() {

assertTrue(m.isTrueVersion2("true"));

assertFalse(m.isTrueVersion2("true2"));

assertTrue(m.isTrueVersion2("True"));;

}

@Test

public void testIsTrueOrYes() {

assertTrue(m.isTrueOrYes("true"));

assertTrue(m.isTrueOrYes("yes"));

assertTrue(m.isTrueOrYes("Yes"));

assertFalse(m.isTrueOrYes("no"));

}

@Test

public void testContainsTrue() {

assertTrue(m.containsTrue("thetruewithin"));

}

@Test

public void testIsThreeLetters() {

assertTrue(m.isThreeLetters("abc"));

assertFalse(m.isThreeLetters("abcd"));

}

@Test

public void testisNoNumberAtBeginning() {

assertTrue(m.isNoNumberAtBeginning("abc"));

assertFalse(m.isNoNumberAtBeginning("1abcd"));

assertTrue(m.isNoNumberAtBeginning("a1bcd"));

assertTrue(m.isNoNumberAtBeginning("asdfdsf"));

}

@Test

public void testisIntersection() {

assertTrue(m.isIntersection("1"));

assertFalse(m.isIntersection("abcksdfkdskfsdfdsf"));

assertTrue(m.isIntersection("skdskfjsmcnxmvjwque484242"));

}

@Test

public void testLessThenThreeHundred() {

assertTrue(m.isLessThenThreeHundred("288"));

assertFalse(m.isLessThenThreeHundred("3288"));

assertFalse(m.isLessThenThreeHundred("328 8"));

assertTrue(m.isLessThenThreeHundred("1"));

assertTrue(m.isLessThenThreeHundred("99"));

assertFalse(m.isLessThenThreeHundred("300"));

}

}

## [5. Pattern and Matcher](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#pattern-and-matcher)

For advanced regular expressions the java.util.regex.Pattern and java.util.regex.Matcher classes are used.

You first create a Pattern object which defines the regular expression. This Pattern object allows you to create a Matcher object for a given string. This Matcher object then allows you to do regex operations on a String.

package de.vogella.regex.test;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexTestPatternMatcher {

public static final String EXAMPLE\_TEST = "This is my small example string which I'm going to use for pattern matching.";

public static void main(String[] args) {

Pattern pattern = Pattern.compile("\\w+");

// in case you would like to ignore case sensitivity,

// you could use this statement:

// Pattern pattern = Pattern.compile("\\s+", Pattern.CASE\_INSENSITIVE);

Matcher matcher = pattern.matcher(EXAMPLE\_TEST);

// check all occurance

while (matcher.find()) {

System.out.print("Start index: " + matcher.start());

System.out.print(" End index: " + matcher.end() + " ");

System.out.println(matcher.group());

}

// now create a new pattern and matcher to replace whitespace with tabs

Pattern replace = Pattern.compile("\\s+");

Matcher matcher2 = replace.matcher(EXAMPLE\_TEST);

System.out.println(matcher2.replaceAll("\t"));

}

}

## [6. Java Regex Examples](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#java-regex-examples)

The following lists typical examples for the usage of regular expressions. I hope you find similarities to your real-world problems.

### [6.1. Or](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#or)

Task: Write a regular expression which matches a text line if this text line contains either the word "Joe" or the word "Jim" or both.

Create a project de.vogella.regex.eitheror and the following class.

package de.vogella.regex.eitheror;

import org.junit.Test;

import static org.junit.Assert.assertFalse;

import static org.junit.Assert.assertTrue;

public class EitherOrCheck {

@Test

public void testSimpleTrue() {

String s = "humbapumpa jim";

assertTrue(s.matches(".\*(jim|joe).\*"));

s = "humbapumpa jom";

assertFalse(s.matches(".\*(jim|joe).\*"));

s = "humbaPumpa joe";

assertTrue(s.matches(".\*(jim|joe).\*"));

s = "humbapumpa joe jim";

assertTrue(s.matches(".\*(jim|joe).\*"));

}

}

### [6.2. Phone number](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#phone-number)

Task: Write a regular expression which matches any phone number.

A phone number in this example consists either out of 7 numbers in a row or out of 3 number, a (white)space or a dash and then 4 numbers.

package de.vogella.regex.phonenumber;

import org.junit.Test;

import static org.junit.Assert.assertFalse;

import static org.junit.Assert.assertTrue;

public class CheckPhone {

@Test

public void testSimpleTrue() {

String pattern = "\\d\\d\\d([,\\s])?\\d\\d\\d\\d";

String s= "1233323322";

assertFalse(s.matches(pattern));

s = "1233323";

assertTrue(s.matches(pattern));

s = "123 3323";

assertTrue(s.matches(pattern));

}

}

### [6.3. Check for a certain number range](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#check-for-a-certain-number-range)

The following example will check if a text contains a number with 3 digits.

Create the Java project de.vogella.regex.numbermatch and the following class.

package de.vogella.regex.numbermatch;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

import org.junit.Test;

import static org.junit.Assert.assertFalse;

import static org.junit.Assert.assertTrue;

public class CheckNumber {

@Test

public void testSimpleTrue() {

String s= "1233";

assertTrue(test(s));

s= "0";

assertFalse(test(s));

s = "29 Kasdkf 2300 Kdsdf";

assertTrue(test(s));

s = "99900234";

assertTrue(test(s));

}

public static boolean test (String s){

Pattern pattern = Pattern.compile("\\d{3}");

Matcher matcher = pattern.matcher(s);

if (matcher.find()){

return true;

}

return false;

}

}

### [6.4. Building a link checker](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#building-a-link-checker)

The following example allows you to extract all valid links from a webpage. It does not consider links which start with "javascript:" or "mailto:".

Create a Java project called de.vogella.regex.weblinks and the following class:

package de.vogella.regex.weblinks;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.net.MalformedURLException;

import java.net.URL;

import java.util.ArrayList;

import java.util.List;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class LinkGetter {

private Pattern htmltag;

private Pattern link;

public LinkGetter() {

htmltag = Pattern.compile("<a\\b[^>]\*href=\"[^>]\*>(.\*?)</a>");

link = Pattern.compile("href=\"[^>]\*\">");

}

public List<String> getLinks(String url) {

List<String> links = new ArrayList<String>();

try {

BufferedReader bufferedReader = new BufferedReader(

new InputStreamReader(new URL(url).openStream()));

String s;

StringBuilder builder = new StringBuilder();

while ((s = bufferedReader.readLine()) != null) {

builder.append(s);

}

Matcher tagmatch = htmltag.matcher(builder.toString());

while (tagmatch.find()) {

Matcher matcher = link.matcher(tagmatch.group());

matcher.find();

String link = matcher.group().replaceFirst("href=\"", "")

.replaceFirst("\">", "")

.replaceFirst("\"[\\s]?target=\"[a-zA-Z\_0-9]\*", "");

if (valid(link)) {

links.add(makeAbsolute(url, link));

}

}

} catch (MalformedURLException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

return links;

}

private boolean valid(String s) {

if (s.matches("javascript:.\*|mailto:.\*")) {

return false;

}

return true;

}

private String makeAbsolute(String url, String link) {

if (link.matches("http://.\*")) {

return link;

}

if (link.matches("/.\*") && url.matches(".\*$[^/]")) {

return url + "/" + link;

}

if (link.matches("[^/].\*") && url.matches(".\*[^/]")) {

return url + "/" + link;

}

if (link.matches("/.\*") && url.matches(".\*[/]")) {

return url + link;

}

if (link.matches("/.\*") && url.matches(".\*[^/]")) {

return url + link;

}

throw new RuntimeException("Cannot make the link absolute. Url: " + url

+ " Link " + link);

}

}

### [6.5. Finding duplicated words](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#finding-duplicated-words)

The following regular expression matches duplicated words.

\b(\w+)\s+\1\b

\b is a word boundary and \1 references to the captured match of the first group, i.e., the first word.

The (?!-in)\b(\w+) \1\b finds duplicate words if they do not start with "-in".

TIP:Add (?s) to search across multiple lines.

### [6.6. Finding elements which start in a new line](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#finding-elements-which-start-in-a-new-line)

The following regular expression allows you to find the "title" word, in case it starts in a new line, potentially with leading spaces.

(\n\s\*)title

### [6.7. Finding (Non-Javadoc) statements](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#finding-non-javadoc-statements)

Sometimes (Non-Javadoc) are used in Java source code to indicate that the method overrides a super method. As of Java 1.6 this can be done via the @Override annotation and it is possible to remove these statements from your code. The following regular expression can be used to identify these statements.

(?s) /\\* \(non-Javadoc\).\*?\\*/

#### [6.7.1. Replacing the DocBook table statement with Asciidoc](http://www.vogella.com/tutorials/JavaRegularExpressions/article.html#replacing-the-docbook-table-statement-with-asciidoc)

You can replace statements like the following:

<programlisting language="java">

<xi:include xmlns:xi="http://www.w3.org/2001/XInclude" parse="text" href="./examples/statements/MyClass.java" />

</programlisting>

Corresponding regex:

`\s+<programlisting language="java">\R.\s+<xi:include xmlns:xi="http://www\.w3\.org/2001/XInclude" parse="text" href="\./examples/(.\*).\s+/>\R.\s+</programlisting>`

Target could be your example:

`\R[source,java]\R----\R include::res/$1[]\R----

Java Regex

Java Regex API provides 1 interface and 3 classes in java.util.regex package.

Matcher class

It implements MatchResult interface. It is a regex engine i.e. used to perform match operations on a character sequence.

No. Method Description

1 boolean matches() test whether the regular expression matches the pattern.

2 boolean find() finds the next expression that matches the pattern.

3 boolean find(int start) finds the next expression that matches the pattern from the given start number.

4 String group() returns the matched subsequence.

5 int start() returns the starting index of the matched subsequence.

6 int end() returns the ending index of the matched subsequence.

7 int groupCount() returns the total number of the matched subsequence.

Pattern class

It is the compiled version of a regular expression. It is used to define a pattern for the regex engine.

No. Method Description

1 static Pattern compile(String regex) compiles the given regex and return the instance of pattern.

2 Matcher matcher(CharSequence input) creates a matcher that matches the given input with pattern.

3 static boolean matches(String regex, CharSequence input) It works as the combination of compile and matcher methods. It compiles the regular expression and matches the given input with the pattern.

4 String[] split(CharSequence input) splits the given input string around matches of given pattern.

5 String pattern() returns the regex pattern.

Example of Java Regular Expressions

There are three ways to write the regex example in java.

import java.util.regex.\*;

public class RegexExample1{

public static void main(String args[]){

//1st way

Pattern p = Pattern.compile(".s");//. represents single character

Matcher m = p.matcher("as");

boolean b = m.matches();

//2nd way

boolean b2=Pattern.compile(".s").matcher("as").matches();

//3rd way

boolean b3 = Pattern.matches(".s", "as");

System.out.println(b+" "+b2+" "+b3);

}}

Test it Now

Output

true true true

Regular Expression . Example

The . (dot) represents a single character.

import java.util.regex.\*;

class RegexExample2{

public static void main(String args[]){

System.out.println(Pattern.matches(".s", "as"));//true (2nd char is s)

System.out.println(Pattern.matches(".s", "mk"));//false (2nd char is not s)

System.out.println(Pattern.matches(".s", "mst"));//false (has more than 2 char)

System.out.println(Pattern.matches(".s", "amms"));//false (has more than 2 char)

System.out.println(Pattern.matches("..s", "mas"));//true (3rd char is s)

}}

Test it Now

Regex Character classes

No. Character Class Description

1 [abc] a, b, or c (simple class)

2 [^abc] Any character except a, b, or c (negation)

3 [a-zA-Z] a through z or A through Z, inclusive (range)

4 [a-d[m-p]] a through d, or m through p: [a-dm-p] (union)

5 [a-z&&[def]] d, e, or f (intersection)

6 [a-z&&[^bc]] a through z, except for b and c: [ad-z] (subtraction)

7 [a-z&&[^m-p]] a through z, and not m through p: [a-lq-z](subtraction)

Regular Expression Character classes Example

import java.util.regex.\*;

class RegexExample3{

public static void main(String args[]){

System.out.println(Pattern.matches("[amn]", "abcd"));//false (not a or m or n)

System.out.println(Pattern.matches("[amn]", "a"));//true (among a or m or n)

System.out.println(Pattern.matches("[amn]", "ammmna"));//false (m and a comes more than once)

}}

Test it Now

Regex Quantifiers

The quantifiers specify the number of occurrences of a character.

Regex Description

X? X occurs once or not at all

X+ X occurs once or more times

X\* X occurs zero or more times

X{n} X occurs n times only

X{n,} X occurs n or more times

X{y,z} X occurs at least y times but less than z times

Regular Expression Character classes and Quantifiers Example

import java.util.regex.\*;

class RegexExample4{

public static void main(String args[]){

System.out.println("? quantifier ....");

System.out.println(Pattern.matches("[amn]?", "a"));//true (a or m or n comes one time)

System.out.println(Pattern.matches("[amn]?", "aaa"));//false (a comes more than one time)

System.out.println(Pattern.matches("[amn]?", "aammmnn"));//false (a m and n comes more than one time)

System.out.println(Pattern.matches("[amn]?", "aazzta"));//false (a comes more than one time)

System.out.println(Pattern.matches("[amn]?", "am"));//false (a or m or n must come one time)

System.out.println("+ quantifier ....");

System.out.println(Pattern.matches("[amn]+", "a"));//true (a or m or n once or more times)

System.out.println(Pattern.matches("[amn]+", "aaa"));//true (a comes more than one time)

System.out.println(Pattern.matches("[amn]+", "aammmnn"));//true (a or m or n comes more than once)

System.out.println(Pattern.matches("[amn]+", "aazzta"));//false (z and t are not matching pattern)

System.out.println("\* quantifier ....");

System.out.println(Pattern.matches("[amn]\*", "ammmna"));//true (a or m or n may come zero or more times)

}}

Test it Now

Regex Metacharacters

The regular expression metacharacters work as a short codes.

Regex Description

. Any character (may or may not match terminator)

\d Any digits, short of [0-9]

\D Any non-digit, short for [^0-9]

\s Any whitespace character, short for [\t\n\x0B\f\r]

\S Any non-whitespace character, short for [^\s]

\w Any word character, short for [a-zA-Z\_0-9]

\W Any non-word character, short for [^\w]

\b A word boundary

\B A non word boundary

Regular Expression Metacharacters Example

import java.util.regex.\*;

class RegexExample5{

public static void main(String args[]){

System.out.println("metacharacters d....");\\d means digit

System.out.println(Pattern.matches("\\d", "abc"));//false (non-digit)

System.out.println(Pattern.matches("\\d", "1"));//true (digit and comes once)

System.out.println(Pattern.matches("\\d", "4443"));//false (digit but comes more than once)

System.out.println(Pattern.matches("\\d", "323abc"));//false (digit and char)

System.out.println("metacharacters D....");\\D means non-digit

System.out.println(Pattern.matches("\\D", "abc"));//false (non-digit but comes more than once)

System.out.println(Pattern.matches("\\D", "1"));//false (digit)

System.out.println(Pattern.matches("\\D", "4443"));//false (digit)

System.out.println(Pattern.matches("\\D", "323abc"));//false (digit and char)

System.out.println(Pattern.matches("\\D", "m"));//true (non-digit and comes once)

System.out.println("metacharacters D with quantifier....");

System.out.println(Pattern.matches("\\D\*", "mak"));//true (non-digit and may come 0 or more times)

}}

Test it Now

Regular Expression Question 1

/\*Create a regular expression that accepts alpha numeric characters only. Its

length must be 6 characters long only.\*/

import java.util.regex.\*;

class RegexExample6{

public static void main(String args[]){

System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun32"));//true

System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "kkvarun32"));//false (more than 6 char)

System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "JA2Uk2"));//true

System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun$2"));//false ($ is not matched)

}}

Test it Now

Regular Expression Question 2

/\*Create a regular expression that accepts 10 digit numeric characters

starting with 7, 8 or 9 only.\*/

import java.util.regex.\*;

class RegexExample7{

public static void main(String args[]){

System.out.println("by character classes and quantifiers ...");

System.out.println(Pattern.matches("[789]{1}[0-9]{9}", "9953038949"));//true

System.out.println(Pattern.matches("[789][0-9]{9}", "9953038949"));//true

System.out.println(Pattern.matches("[789][0-9]{9}", "99530389490"));//false (11 characters)

System.out.println(Pattern.matches("[789][0-9]{9}", "6953038949"));//false (starts from 6)

System.out.println(Pattern.matches("[789][0-9]{9}", "8853038949"));//true

System.out.println("by metacharacters ...");

System.out.println(Pattern.matches("[789]{1}\\d{9}", "8853038949"));//true

System.out.println(Pattern.matches("[789]{1}\\d{9}", "3853038949"));//false (starts from 3)

}}

Test it Now

Java Regex Finder Example

import java.util.regex.Pattern;

import java.util.Scanner;

import java.util.regex.Matcher;

public class RegexExample8{

public static void main(String[] args){

Scanner sc=new Scanner(System.in);

while (true) {

System.out.println("Enter regex pattern:");

Pattern pattern = Pattern.compile(sc.nextLine());

System.out.println("Enter text:");

Matcher matcher = pattern.matcher(sc.nextLine());

boolean found = false;

while (matcher.find()) {

System.out.println("I found the text "+matcher.group()+" starting at index "+

matcher.start()+" and ending at index "+matcher.end());

found = true;

}

if(!found){

System.out.println("No match found.");

}

}

}

}

Output:

Enter regex pattern: java

Enter text: this is java, do you know java

I found the text java starting at index 8 and ending at index 12

I found the text java starting at index 26 and ending at index 3

java.util.regex

## Class Pattern

* [java.lang.Object](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html)
  + java.util.regex.Pattern
* All Implemented Interfaces:

[Serializable](https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)

public final class **Pattern**

extends [Object](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html)

implements [Serializable](https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)

A compiled representation of a regular expression.

A regular expression, specified as a string, must first be compiled into an instance of this class. The resulting pattern can then be used to create a [Matcher](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html) object that can match arbitrary [character sequences](https://docs.oracle.com/javase/7/docs/api/java/lang/CharSequence.html) against the regular expression. All of the state involved in performing a match resides in the matcher, so many matchers can share the same pattern.

A typical invocation sequence is thus

Pattern p = Pattern.[compile](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#compile%28java.lang.String%29)("a\*b");

Matcher m = p.[matcher](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#matcher%28java.lang.CharSequence%29)("aaaaab");

boolean b = m.[matches](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html#matches%28%29)();

A [matches](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#matches%28java.lang.String,%20java.lang.CharSequence%29) method is defined by this class as a convenience for when a regular expression is used just once. This method compiles an expression and matches an input sequence against it in a single invocation. The statement

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

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

Instances of this class are immutable and are safe for use by multiple concurrent threads. Instances of the [Matcher](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html) class are not safe for such use.

#### Summary of regular-expression constructs

\*

|  |  |
| --- | --- |
| **Construct** | **Matches** |
|  |  |
| **Characters** | |
| *x* | The character *x* |
| \\ | The backslash character |
| \0*n* | The character with octal value 0*n* (0 <= *n* <= 7) |
| \0*nn* | The character with octal value 0*nn* (0 <= *n* <= 7) |
| \0*mnn* | The character with octal value 0*mnn* (0 <= *m* <= 3, 0 <= *n* <= 7) |
| \x*hh* | The character with hexadecimal value 0x*hh* |
| \u*hhhh* | The character with hexadecimal value 0x*hhhh* |
| \x*{h...h}* | The character with hexadecimal value 0x*h...h* ([Character.MIN\_CODE\_POINT](https://docs.oracle.com/javase/7/docs/api/java/lang/Character.html" \l "MIN_CODE_POINT)  <= 0x*h...h* <=  [Character.MAX\_CODE\_POINT](https://docs.oracle.com/javase/7/docs/api/java/lang/Character.html#MAX_CODE_POINT)) |
| \t | The tab character ('\u0009') |
| \n | The newline (line feed) character ('\u000A') |
| \r | The carriage-return character ('\u000D') |
| \f | The form-feed character ('\u000C') |
| \a | The alert (bell) character ('\u0007') |
| \e | The escape character ('\u001B') |
| \c*x* | The control character corresponding to *x* |
|  |  |
| **Character classes** | |
| [abc] | a, b, or c (simple class) |
| [^abc] | Any character except a, b, or c (negation) |
| [a-zA-Z] | a through z or A through Z, inclusive (range) |
| [a-d[m-p]] | a through d, or m through p: [a-dm-p] (union) |
| [a-z&&[def]] | d, e, or f (intersection) |
| [a-z&&[^bc]] | a through z, except for b and c: [ad-z] (subtraction) |
| [a-z&&[^m-p]] | a through z, and not m through p: [a-lq-z](subtraction) |
|  |  |
| **Predefined character classes** | |
| . | Any character (may or may not match [line terminators](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#lt)) |
| \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] |
|  |  |
| **POSIX character classes (US-ASCII only)** | |
| \p{Lower} | A lower-case alphabetic character: [a-z] |
| \p{Upper} | An upper-case alphabetic character:[A-Z] |
| \p{ASCII} | All ASCII:[\x00-\x7F] |
| \p{Alpha} | An alphabetic character:[\p{Lower}\p{Upper}] |
| \p{Digit} | A decimal digit: [0-9] |
| \p{Alnum} | An alphanumeric character:[\p{Alpha}\p{Digit}] |
| \p{Punct} | Punctuation: One of !"#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~ |
| \p{Graph} | A visible character: [\p{Alnum}\p{Punct}] |
| \p{Print} | A printable character: [\p{Graph}\x20] |
| \p{Blank} | A space or a tab: [ \t] |
| \p{Cntrl} | A control character: [\x00-\x1F\x7F] |
| \p{XDigit} | A hexadecimal digit: [0-9a-fA-F] |
| \p{Space} | A whitespace character: [ \t\n\x0B\f\r] |
|  |  |
| **java.lang.Character classes (simple** [**java character type**](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#jcc)**)** | |
| \p{javaLowerCase} | Equivalent to java.lang.Character.isLowerCase() |
| \p{javaUpperCase} | Equivalent to java.lang.Character.isUpperCase() |
| \p{javaWhitespace} | Equivalent to java.lang.Character.isWhitespace() |
| \p{javaMirrored} | Equivalent to java.lang.Character.isMirrored() |
|  |  |
| **Classes for Unicode scripts, blocks, categories and binary properties** | |
| \p{IsLatin} | A Latin script character ([script](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#usc)) |
| \p{InGreek} | A character in the Greek block ([block](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#ubc)) |
| \p{Lu} | An uppercase letter ([category](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#ucc)) |
| \p{IsAlphabetic} | An alphabetic character ([binary property](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#ubpc)) |
| \p{Sc} | A currency symbol |
| \P{InGreek} | Any character except one in the Greek block (negation) |
| [\p{L}&&[^\p{Lu}]] | Any letter except an uppercase letter (subtraction) |
|  |  |
| **Boundary matchers** | |
| ^ | The beginning of a line |
| $ | The end of a line |
| \b | A word boundary |
| \B | 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](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#lt), if any |
| \z | The end of the input |
|  |  |
| **Greedy quantifiers** | |
| *X*? | *X*, once or not at all |
| *X*\* | *X*, zero or more times |
| *X*+ | *X*, one or more times |
| *X*{*n*} | *X*, exactly *n* times |
| *X*{*n*,} | *X*, at least *n* times |
| *X*{*n*,*m*} | *X*, at least *n* but not more than *m* times |
|  |  |
| **Reluctant quantifiers** | |
| *X*?? | *X*, once or not at all |
| *X*\*? | *X*, zero or more times |
| *X*+? | *X*, one or more times |
| *X*{*n*}? | *X*, exactly *n* times |
| *X*{*n*,}? | *X*, at least *n* times |
| *X*{*n*,*m*}? | *X*, at least *n* but not more than *m* times |
|  |  |
| **Possessive quantifiers** | |
| *X*?+ | *X*, once or not at all |
| *X*\*+ | *X*, zero or more times |
| *X*++ | *X*, one or more times |
| *X*{*n*}+ | *X*, exactly *n* times |
| *X*{*n*,}+ | *X*, at least *n* times |
| *X*{*n*,*m*}+ | *X*, at least *n* but not more than *m* times |
|  |  |
| **Logical operators** | |
| *XY* | *X* followed by *Y* |
| *X*|*Y* | Either *X* or *Y* |
| (*X*) | X, as a [capturing group](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#cg) |
|  |  |
| **Back references** | |
| \*n* | Whatever the *n*th [capturing group](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#cg) matched |
| \*k*<*name*> | Whatever the [named-capturing group](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#groupname) "name" matched |
|  |  |
| **Quotation** | |
| \ | Nothing, but quotes the following character |
| \Q | Nothing, but quotes all characters until \E |
| \E | Nothing, but ends quoting started by \Q |
|  |  |
| **Special constructs (named-capturing and non-capturing)** | |
| (?<[name](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#groupname)>*X*) | *X*, as a named-capturing group |
| (?:*X*) | *X*, as a non-capturing group |
| (?idmsuxU-idmsuxU) | Nothing, but turns match flags [i](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#CASE_INSENSITIVE) [d](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#UNIX_LINES) [m](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#MULTILINE) [s](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#DOTALL) [u](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#UNICODE_CASE) [x](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#COMMENTS) [U](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#UNICODE_CHARACTER_CLASS) on - off |
| (?idmsux-idmsux:*X*) | *X*, as a [non-capturing group](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#cg) with the given flags [i](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#CASE_INSENSITIVE) [d](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#UNIX_LINES) [m](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#MULTILINE) [s](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#DOTALL) [u](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#UNICODE_CASE) [x](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#COMMENTS) on - off |
| (?=*X*) | *X*, via zero-width positive lookahead |
| (?!*X*) | *X*, via zero-width negative lookahead |
| (?<=*X*) | *X*, via zero-width positive lookbehind |
| (?<!*X*) | *X*, via zero-width negative lookbehind |
| (?>*X*) | *X*, as an independent, non-capturing group |

#### Backslashes, escapes, and quoting

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

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

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

#### Character Classes

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

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

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

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

#### Line terminators

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

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

If [UNIX\_LINES](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html" \l "UNIX_LINES) mode is activated, then the only line terminators recognized are newline characters.

The regular expression . matches any character except a line terminator unless the [DOTALL](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#DOTALL) flag is specified.

By default, the regular expressions ^ and $ ignore line terminators and only match at the beginning and the end, respectively, of the entire input sequence. If [MULTILINE](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#MULTILINE) mode is activated then ^ matches at the beginning of input and after any line terminator except at the end of input. When in [MULTILINE](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#MULTILINE) mode $ matches just before a line terminator or the end of the input sequence.

#### Groups and capturing

##### Group number

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

|  |  |
| --- | --- |
| **1** | ((A)(B(C))) |
| **2** | (A) |
| **3** | (B(C)) |
| **4** | (C) |

Group zero always stands for the entire expression.

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

##### Group name

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

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

A named-capturing group is still numbered as described in [Group number](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html" \l "gnumber).

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

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

#### Unicode support

This class is in conformance with Level 1 of [*Unicode Technical Standard #18: Unicode Regular Expression*](http://www.unicode.org/reports/tr18/), plus RL2.1 Canonical Equivalents.

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

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

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

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

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

The script names supported by Pattern are the valid script names accepted and defined by [UnicodeScript.forName](https://docs.oracle.com/javase/7/docs/api/java/lang/Character.UnicodeScript.html" \l "forName%28java.lang.String%29).

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

The block names supported by Pattern are the valid block names accepted and defined by [UnicodeBlock.forName](https://docs.oracle.com/javase/7/docs/api/java/lang/Character.UnicodeBlock.html" \l "forName%28java.lang.String%29).

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

The supported categories are those of *[The Unicode Standard](http://www.unicode.org/unicode/standard/standard.html)* in the version specified by the [Character](https://docs.oracle.com/javase/7/docs/api/java/lang/Character.html) class. The category names are those defined in the Standard, both normative and informative.

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

* + Alphabetic
  + Ideographic
  + Letter
  + Lowercase
  + Uppercase
  + Titlecase
  + Punctuation
  + Control
  + White\_Space
  + Digit
  + Hex\_Digit
  + Noncharacter\_Code\_Point
  + Assigned

**Predefined Character classes** and **POSIX character classes** are in conformance with the recommendation of *Annex C: Compatibility Properties* of *[Unicode Regular Expression](http://www.unicode.org/reports/tr18/)* , when [UNICODE\_CHARACTER\_CLASS](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#UNICODE_CHARACTER_CLASS) flag is specified.

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

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

#### Comparison to Perl 5

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

Perl constructs not supported by this class:

* + Predefined character classes (Unicode character)

\h    A horizontal whitespace

\H    A non horizontal whitespace

\v    A vertical whitespace

\V    A non vertical whitespace

\R    Any Unicode linebreak sequence \u000D\u000A|[\u000A\u000B\u000C\u000D\u0085\u2028\u2029]

\X    Match Unicode *[extended grapheme cluster](http://www.unicode.org/reports/tr18/" \l "Default_Grapheme_Clusters)*

* + The backreference constructs, \g{*n*} for the *n*th[capturing group](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#cg) and \g{*name*} for [named-capturing group](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#groupname).
  + The named character construct, \N{*name*} for a Unicode character by its name.
  + The conditional constructs (?(*condition*)*X*) and (?(*condition*)*X*|*Y*),
  + The embedded code constructs (?{*code*}) and (??{*code*}),
  + The embedded comment syntax (?#comment), and
  + The preprocessing operations \l \u, \L, and \U.

Constructs supported by this class but not by Perl:

* + Character-class union and intersection as described [above](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#cc).

Notable differences from Perl:

* + In Perl, \1 through \9 are always interpreted as back references; a backslash-escaped number greater than 9 is treated as a back reference if at least that many subexpressions exist, otherwise it is interpreted, if possible, as an octal escape. In this class octal escapes must always begin with a zero. In this class, \1 through \9 are always interpreted as back references, and a larger number is accepted as a back reference if at least that many subexpressions exist at that point in the regular expression, otherwise the parser will drop digits until the number is smaller or equal to the existing number of groups or it is one digit.
  + Perl uses the g flag to request a match that resumes where the last match left off. This functionality is provided implicitly by the [Matcher](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html) class: Repeated invocations of the [find](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html#find%28%29) method will resume where the last match left off, unless the matcher is reset.
  + In Perl, embedded flags at the top level of an expression affect the whole expression. In this class, embedded flags always take effect at the point at which they appear, whether they are at the top level or within a group; in the latter case, flags are restored at the end of the group just as in Perl.

For a more precise description of the behavior of regular expression constructs, please see [*Mastering Regular Expressions, 3nd Edition*, Jeffrey E. F. Friedl, O'Reilly and Associates, 2006.](http://www.oreilly.com/catalog/regex3/)

**See Also:**

[String.split(String, int)](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#split%28java.lang.String,%20int%29), [String.split(String)](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#split%28java.lang.String%29), [Serialized Form](https://docs.oracle.com/javase/7/docs/api/serialized-form.html#java.util.regex.Pattern)