This material has been collected from the numerous sources such as StackOverfloow including my own and my students over the years of teaching. Please help me to keep this tutorial up-to-date by reporting any issues or questions. Please send any comments or criticisms to [idebtor@gmail.com](mailto:idebtor@gmail.com). Your assistances and comments will be appreciated.

Regex Version 0.1

Contents

[Regex: Regular Expression – 정규식 1](#_Toc497394053)

[For Your References 2](#_Toc497394054)

[Regular Expressions by Example 2](#_Toc497394055)

[Honor code and Submitting your solution 5](#_Toc497394056)

[Files to submit 5](#_Toc497394057)

[Due and Grade points 5](#_Toc497394058)

# Regex: Regular Expression – 정규식

Using a regular expression, complete the following Java program that tests whether an input string is a hexadecimal string or not.

* Use for-each loop
* Use split() method and trim() method as needed
* Use the exact strings defined below.
* **Pay attention that there are some extra spaces or no spaces between hex numbers or in the beginning of the string.**

public class Regex {

public static boolean isHexString(String hexString) {

return hexString.matches("[0-9A-F]"); // fix this regular expression

}

public static void main(String[] args) {

String str0 = "1F";

System.out.println(str0 + "\t = " + isHexString(str0));

String str1 = "0x1D9, 0x20, 0xaBC";

// your code here

String str2 = "0x, 0X0X X, abg, ab";

// your code here

String str3 = " 123, aBC, a5,0XFF, 0xF, 0 ";

// your code here

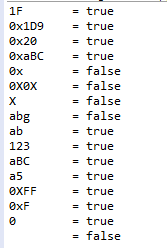
String blank = "";

System.out.println(blank + "\t = " + isHexString(blank));

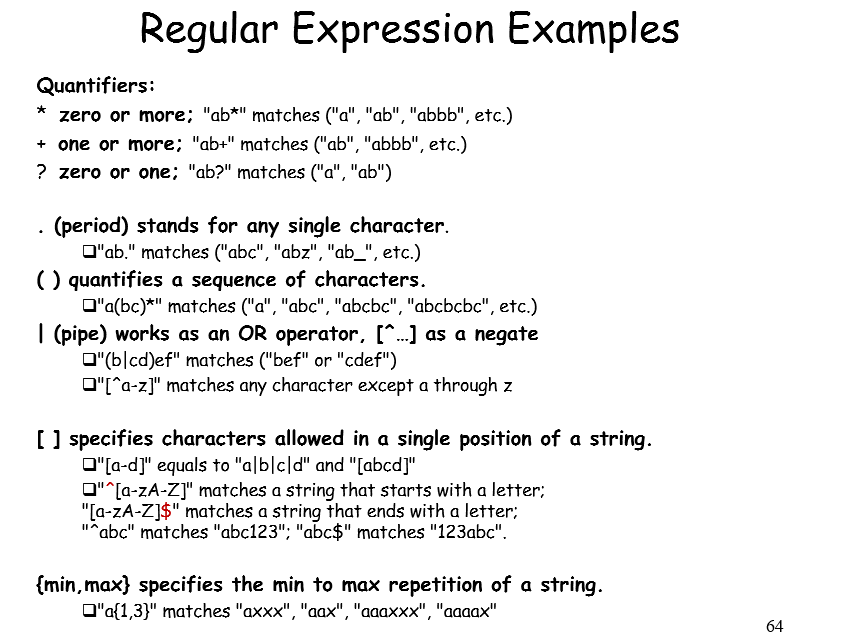
}

}

The expected output is as shown below:



# For Your References



# 

# Regular Expressions by Example

Main Sources: msdn.microsoft.com, [www.phpbuilder.com/columns/dario19990616.php3?page=1](http://www.phpbuilder.com/columns/dario19990616.php3?page=1)

You may use regexr.com to test regular expressions interactively.

Regular expressions provide a powerful, flexible, and efficient method for processing text. The extensive pattern-matching notation of regular expressions allows you to quickly parse large amounts of text to find specific character patterns; to extract, edit, replace, or delete text substrings; or to add the extracted strings to a collection in order to generate a report. For many applications that deal with strings, regular expressions are an indispensable tool.

**Basic Rules**

Let's take a look at two special symbols(**anchors**): '^' (caret) and '$'. What they do is to indicate the *start* and the *end* of a string, respectively, like this:

"^The": matches any string that starts with "The";   
"of despair$": matches a string that ends in the substring "of despair";   
"^abc$": a string that starts and ends with "abc" -- that could only be "abc" itself!   
"notice": a string that has the text "notice" in it.

You can see that if you don't use either of the two characters we mentioned, as in the last example, you're saying that the pattern may occur anywhere inside the string -- you're not "hooking" it to any of the edges.

There are also the symbols(**quantifiers**) '\*', '+', and '?', which denote the number of times a character or a sequence of characters may occur. What they mean is: "zero or more", "one or more", and "zero or one." Here are some examples:

"ab\*": matches a string that has an *a* followed by zero or more *b*'s ("a", "ab", "abbb", etc.);   
"ab+": same, but there's at least one *b* ("ab", "abbb", etc.);   
"ab?": there might be a *b* or not;   
"a?b+$": a possible *a* followed by one or more *b*'s ending a string.

You can also use *bounds*, which come inside braces and indicate ranges in the number of occurrences:

"ab{2}": matches a string that has an *a* followed by exactly two *b*'s ("abb");   
"ab{2,}": there are at least two *b*'s ("abb", "abbbb", etc.);   
"ab{3,5}": from three to five *b*'s ("abbb", "abbbb", or "abbbbb").

***Note*** that you must always specify the first number of a range (i.e, "{0,2}", not "{,2}"). Also, as you might have noticed, the symbols '\*', '+', and '?' have the same effect as using the **bounds** "{0,}", "{1,}", and "{0,1}", respectively.

Now, to quantify **a sequence of characters**, put them inside parentheses:

"a(bc)\*": matches a string that has an *a* followed by zero or more copies of the sequence "bc";   
"a(bc){1,5}": one through five copies of "bc."

There's also the '|' symbol, which works as an **OR operator**:

"hi|hello": matches a string that has either "hi" or "hello" in it;   
"(b|cd)ef": a string that has either "bef" or "cdef";   
"(a|b)\*c": a string that has a sequence of alternating *a*'s and *b*'s ending in a *c*;

A period ('.') stands for **any single character**:

"a.[0-9]": matches a string that has an *a* followed by one character and a digit;   
"^.{3}$": a string with exactly 3 characters.

***Bracket expressions*** specify which characters are allowed in **a single position of a string:**

"[ab]": matches a string that has either an *a* or a *b* (that's the same as "a|b");   
"[a-d]": a string that has lowercase letters 'a' through 'd' (that's equal to "a|b|c|d" and even "[abcd]");   
"^[a-zA-Z]": a string that starts with a letter;   
"[0-9]%": a string that has a single digit before a percent sign;   
",[a-zA-Z0-9]$": a string that ends in a comma followed by **an alphanumeric character**.

You can also list which characters you DON'T want -- just **use a '^' as the first symbol in a bracket expression** (i.e., "%[^a-zA-Z]%" matches a string with a character that is not a letter between two percent signs).

In order to be taken literally, you must escape the characters "^.[$()|\*+?{\" with a backslash ('\'), as they have special meaning. On top of that, you must escape the backslash character itself in PHP3 strings, so, for instance, the regular expression "(\$|?[0-9]+" would have the function call: ereg("(\\$|?[0-9]+", $str) (what string does that validate?).

**\d matches a single character that is a digit, \w matches a "word character" (alphanumeric characters plus underscore), and \s matches a whitespace character (includes tabs and line breaks).**

Just don't forget that bracket expressions are an exception to that rule--inside them, all special characters, including the backslash ('\'), lose their special powers (i.e., "[\*\+?{}.]" matches exactly any of the characters inside the brackets). And, as the regex UNIX man pages tell us: "To include a literal ']' in the list, make it the first character (following a possible '^'). To include a literal '-', make it the first or last character, or the second endpoint of a range."

**Validating E-mail Addresses**

Ok, let's take on e-mail addresses. There are three parts in an e-mail address: the POP3 user name (everything to the left of the '@'), the '@', and the server name (the rest). The user name may contain upper or lowercase letters, digits, periods ('.'), minus signs ('-'), and underscore signs ('\_'). That's also the case for the server name, except for underscore signs, which may not occur.

Now, you can't start or end a user name with a period, it doesn't seem reasonable. The same goes for the domain name. And you can't have two consecutive periods, there should be at least one other character between them. Let's see how we would write an expression to validate the user name part:

^[\_a-zA-Z0-9-]+$

That doesn't allow a period yet. Let's change it:

^[\_a-zA-Z0-9-]+(\.[\_a-zA-Z0-9-]+)\*$

That says: "at least one valid character followed by zero or more sets consisting of a period and one or more valid characters."

To simplify things a bit, we can use the expression above with eregi(), instead of ereg(). Because eregi() is not sensitive to case, we don't have to specify both ranges "a-z" and "A-Z" -- one of them is enough:

^[\_a-z0-9-]+(\.[\_a-z0-9-]+)\*$

For the server name it's the same, but without the underscores:

^[a-z0-9-]+(\.[a-z0-9-]+)\*$

Done. Now, joining both expressions around the 'at' sign, we get:

^[\_a-z0-9-]+(\.[\_a-z0-9-]+)\*@[a-z0-9-]+(\.[a-z0-9-]+)\*$

**Replacing**

Suppose we want to separate all words in a string by commas. Then, use Regex.Replace().

Regex.Replace (strin, "[ \n\r\t]+", ","); // There is a space between “[ and \n.

**Some exercises -** Now here's something to make you busy.

* What does this do? Regex.Replace (string, "[^\w\.%@-]", "").
* Make up a regular expression to check a hexadecimal number: 5F, 5f, 0x5F, 0X5A.
* Make up a regular expression to convert UNIX text file into Windows text file.
* Make up a regular expression to extract IP addresses.
* Modify our e-mail-validating regular expression to force the server name part to consist of at least two names (hint: only one character needs to be changed);  
   ^[\_a-z0-9-]+(\.[\_a-z0-9-]+)\*@[a-z0-9-]+(\.[a-z0-9-]+)\*$
* Make up a regular expression in a short form using \w for ^[\_a-z0-9-]+(\.[\_a-z0-9-]+)\* .
* How about extracting dates(11/03/05, 11/3/2005, 2005/11/3), social security numbers (800610-2030405), and cell phone numbers(010-537-2819, 010.1234.5678, 011 537-2819)?

# Honor code and Submitting your solution

* This is not a group project. But I encourage you to share your knowledge or to get help each other. You may discuss or debug the code together, but never share your code with others. You are responsible for submitting your work by yourself.
* Include the following line at the top of your source files with your name signed.  
  On my honour, I pledge that I have neither received nor provided improper assistance in the completion of this assignment. Signed: \_\_\_\_\_\_\_\_\_\_\_\_\_
* Include your name, student number and email address.  
  Include your group partner's name and student number.
* Make sure your code **compiles** and **runs** right before you submit it. Every semester, we get dozens of submissions that don't even compile. Don't make "a tiny last-minute change" and assume your code still compiles. You will not receive sympathy for code that "almost" works.
* If you only manage to work out the homework problem partially before the due, you still need to turn it in. However, don’t turn it in if it does not compile and run.
* After submitting, if you realize one of your programs is flawed, you may fix it and submit again as long as it is **before the deadline**. You will have to resubmit any related files together, even if you only change one. You may submit as often as you like. **Only the last version** you submit before the deadline will be graded.

## Files to submit

Submit the following source file on time in **hw6** folder in Piazza

* + Regex.java

## Due and Grade points

* Due: 10:00 am, Monday, June 6, 2017
* Grade points: 3