Regular Expressions

A **regular expression** (regex) is a special text string for describing a search pattern. A regular expression works by matching a string against a template or pattern, and in its simplest form, returning a Boolean to say "yes, the string does look like the pattern" or "no, that doesn't match".

Introduction

Let's start simple. Suppose you want to search for a string with the word "cat" in it; your regular expression would simply by "cat". If you search is case-insensitive, the words "catalog", "Catherine", or "sophisticated" would also match:

Regular expression: cat **Matches**: cat, catalog, Catherine, sophisticated

The period notation

Imagine you are playing Scrabble and need a three-letter word starting with the letter "t" and end with the letter "n". Imagine also that you have an English dictionary and will search through its entire contents for a match using a regular expression. To search for such a regular expression, you would use a **wildcard notation** – the period (.) character. The regular expression would then be "t.n" and would match "tan", "Ten", "tin", and "ton"; it would also match "t#n", "tpn", and even "t n". This is because the period character matches everything, including spaces, the tab character, and even line breaks:

Regular expression: t.n **Matches**: tan, Ten, tin, ton, t n, t#n, tpn, etc.

The bracket notation

To solve the problem of the period's indiscriminate matches, you can specify characters you consider meaningful with bracket ("[]") expression, so that only those characters would match the regular expression. Thus, "t[aeio]n" would just match "tan", "Ten", "tin", and "ton". "Toon" would not match because you can only match a single character within the bracket notation:

Regular expression: t[aeio]n **Matches**: tan, Ten, tin, ton

The OR operator

If you want to match "toon" in addition to all the words matched in the previous section you can use the "|" notation, which is basically an OR operator. To match "toon", use the regular expression "t(a|e|i|o|oo)n". You cannot use the bracket notation here because it will only match a single character. Instead, use parentheses – "( )". You can also use parentheses for groupings (more on that later):

Regular expression: t(a|e|i|o|oo)n
Matches: tan, Ten, tin, ton, toon

The quantifier notations

The table below shows the quantifier notations used to determine how many times a given notation to the immediate left of the quantifier notation should repeat itself:

Quantifier Notations

Notation	Number of Times
*	0 or more times
+	1 or more times
?	0 or 1 time
{n}	Exactly n number of times
{n,m}	n to m number of times

Let's say you want to search for a social security number in a text file. The format for US social security numbers is 999-99-9999. The regular expression you would use to match this is shown below. In regular expressions, the hyphen ("-") notation has special meaning; it indicates a range that would match any number from 0 to 9. As a result, you must escape the "-" character with a forward slash("\") when matching the literal hyphens in a social security number. Since the forward slash("\") has special meaning (used to invoke an escape sequence), you must escape it as well.

Regular expression : [0-9]{3} \\- [0-9]{2} \\- [0-9]{4}
The first 3 digits The next 2 digits The last 4 digits

Matches: All social security numbers of the form 123-12-1234

If, in your search, you wish to make the hyphens optional – if, say, you consider both 999-99-9999 and 9999999999 acceptable formats – you can use the "?" quantifier notation.

Optional hyphen Optional hyphen
Regular expression: [0-9]{3} \\-? [0-9]{2} \\-? [0-9]{4}
The first 3 digits The next 2 digits The last 4 digits

Matches: All social security numbers of the forms 123-12-1234 and 123121234

Let's take a look at another example. One format for US car plate numbers consists of four numeric characters followed by two letters. The regular expression first comprises the numeric part, "[0-9]{4}", followed by the textual part, "[A-Z]{2}".

Regular expression: [0-9]{4} [A-Z]{2}
The first 4 digits The last 2 alphabets

Matches: Typical US car plate numbers, such as 8836KV

The NOT notation

The "^" notation is also called the NOT notation. If used in brackets, "^" indicates the character you **don't** want to match. For example, the expression below matches all words except those starting with the letter x.

Regular expression: `[^x]` `[a-z]+`

`[^x]`	`[a-z]+`
The first character must not be 'x'	The subsequent characters can by anything from a-z

Matches: All words except those that start with the letter x

The parentheses and space notations

Say you're trying to extract the birth month from a person's birth date. The typical birth date is in the following format: June 10, 1968. The regular expression to match the string would be like the one shown below.

Mandatory space		Mandatory comma	Year field, up to 4 digits

Regular expression: `[a-z]+ \\s+ [0-9]{1,2} , \\s* [0-9]{4}`

Matches: All dates with the format Month DD, YYYY

The "\s" notation is the space notation and matches all blank spaces, including tabs.

Other miscellaneous notations

To make life easier, some shorthand notations for commonly used regular expressions have been created, as shown in the table below.

Commonly used Notations

Notation	Equivalent Notation	Explanation
\d	[0-9]	A digit
\D	[^0-9]	A non-digit
\w	[A-Z0-9]	A word character (alphanumeric)
\W	[^A-Z0-9]	A non-word character
\s	[\t\n\r\f]	A whitespace character
\S	[^\t\n\r\f]	A non-word character

String split method

```
public class StringSplit
{
   public static void main(String[] args)
   {
      // Example 1 - delimeter is a space
      // Note - the delimeter is not included in array
      String s1 = "Mary had a little lamb";
      String[] array1 = s1.split(" ");
      for(String s: array1)
      {
         System.out.println(s);
      }
      System.out.println();

      // Example 2 - delimeter is a comma
      String s2 = "Monday,Tuesday,Wednesday,Thursday,Friday";
      String[] array2 = s2.split(",");
      for(String s: array2)
      {
         System.out.println(s);
      }
      System.out.println();

      // Example 3 - delimeter is regular expression (Regex)
      String s3 = "121122211121121";
      String[] array3 = s3.split("2+");  // split at one or more 2's
      for(String s: array3)
      {
         System.out.println(s);
      }
      System.out.println();

      // Example 4 - delimeter is regular expression (Regex)
      String s4 = "125432225432115412";
      String[] array4 = s4.split("[1-3]+");  // split at any number of
                                             //  1's, 2's, or 3's
      for(String s: array4)
      {
         System.out.println(s);
      }
      System.out.println();
   }
}
```

```
Example 1
--------------
Mary
had
a
little
lamb


Example 2
---------------
Monday
Tuesday
Wednesday
Thursday
Friday

Example 3
---------------
1
11
111
11
1

Example 4
----------------
54
54
54

Press any key to continue...
```