**RegexExample1.java**

The first example is actually a basic example

**import java.util.regex.Pattern;**

**import java.util.regex.Matcher;**

**class MatcherExample**

**{**

**public static void main(String args[])**

**{**

**String text="This is the text to be searched for http://pattern.";**

**String patternString=".\*http://.\*";**

**/\***

**\*If any string contains http:// at any position, patternString will find a possible match with it**

**\*as we used .\* in the beginning**

**\*as well as in the end\*/**

**Pattern pattern=Pattern.compile(patternString);**

**/\*Now, Pattern class or java.util.regex.Pattern does not have any constructor. It has static methods to create pattern.**

**One of the function is compile\*/**

**Matcher matcher = pattern.matcher(text);**

**boolean matches=matcher.matches();**

**System.out.println("The pattern is "+((matches==true)?"":"not ")+"found in the text");**

**}**

**}**

It does not show much except the basic use of java.util.regex.Pattern and java.util.regex.Matcher class.

Now, as it can be seen, a pattern string is defined as “**.\*http://.\*”.**

And, the text will be tried to match with the pattern. If the text defined matches with the pattern, it will be printed that “The pattern is found in the text”. And, if the text is not matched with the pattern, it will be printed that “The pattern is not found in the text”.

RegexExample2.java

The next example actually shows us how to define a proper pattern.

Consider this example:

**import java.util.regex.Pattern;**

**import java.util.regex.Matcher;**

**class MatcherExample2**

**{**

**public static void main(String args[])**

**{**

**String text1="The string will be searched for http://pattern";**

**//text1 will be the string which will be searched for "http://pattern" at the end**

**String patternToBeSearched="http://pattern$";**

**Pattern pattern=Pattern.compile(patternToBeSearched);**

**Matcher matcher=pattern.matcher(text1);**

**Boolean foundFlag=matcher.matches();**

**System.out.println("The pattern is "+((foundFlag==true)?"":"not ")+"found in the text");**

**}**

**}**

Now, The intention was to define a pattern with which the String text1 will be matched. Now, the pattern is defined as **"http://pattern$".** Now,

First assumption is that a string which contains “http:pattern” at the end, will be matched with the pattern defined. Hell, no. Because, the pattern is defined in a wrong manner. According to the pattern no character is allowed before the beginning of <http://pattern.> So, the result of the program would be **“The pattern is not found in the text”.**

However, if the String text1 was defined like “http://pattern” the output would be **“The pattern is found in the text”.**

**RegexExample3.java**

Third example is pretty simple. It shows that, if the pattern is defined as “.\*http://pattern$”, the string text1 will not match with the pattern. And the string text2 will be matched with the pattern.

**import java.util.regex.Pattern;**

**import java.util.regex.Matcher;**

**class MatcherExample3**

**{**

**public static void main(String args[])**

**{**

**String text1="The String will be searched for http://pattern at the end";**

**String text2="The String will be searched for http://pattern";**

**String patternToBeSearched=".\*http://pattern$";**

**Pattern pattern=Pattern.compile(patternToBeSearched);**

**Matcher stringToBeChecked1=pattern.matcher(text1);**

**//Now, matcher is the function which tries to match the patternToBeSearched indicated by pattern with the text text1**

**boolean foundFlag=stringToBeChecked1.matches();**

**System.out.println("The pattern is "+((foundFlag==true)?"":"not ")+"found in the text");**

**Matcher stringToBeChecked2=pattern.matcher(text2);**

**foundFlag=stringToBeChecked2.matches();**

**System.out.println("The pattern is "+((foundFlag==true)?"":"not ")+"found in the text");**

**}**

**}**

**RegexExample4.java**

**import java.util.regex.Pattern;**

**import java.util.regex.Matcher;**

**class PatternMatcher4**

**{**

**public static void main(String args[])**

**{**

**String text="http://pattern is meant to be found at the beginning of the string";**

**String patternToBeSearched="^http://pattern.\*";**

**Pattern pattern=Pattern.compile(patternToBeSearched);**

**Matcher matcher=pattern.matcher(text);**

**//this tries to match the pattern in the text**

**boolean matchfoundflag=matcher.matches();**

**System.out.println("The pattern is "+ ((matchfoundflag==true)?"":"not ")+"found in the text");**

**}**

**}**

Now, a pattern is defined In such a way that a string containing <http://pattern> at the very beginning, will match will the pattern.

**RegexExample5.java**

**import java.util.regex.Pattern;**

**import java.util.regex.Matcher;**

**class MatcherExample5**

**{**

**public static void main(String args[])**

**{**

**String text="This string is ended with newline\n";**

**String patternToBeMatched="(.|\n)\*";**

**Pattern pattern=Pattern.compile(patternToBeMatched);**

**Matcher matcher=pattern.matcher(text);**

**boolean foundflag=matcher.matches();**

**System.out.println("The pattern is "+((foundflag==true)?"":"not ")+"found in the text");**

**}**

**}**

Now, if the text is ="This string is ended with newline" with no actual newline and patternToBeMatched is =".\*", a match is found.

However, if the text is="The string is ended with newline\n" with an actual newline and patternToBeMatched is=".\*", no match is found.

Now, if the text is="This string is ended with newline\n"

and patternToBeMatched is="(.|\n)\*" a match is found.

|:- this is or operator. A|B means matches either A or B.

**RegexExample6.java**

**import java.util.regex.Pattern;**

**import java.util.regex.Matcher;**

**class MatcherExample6**

**{**

**public static void main(String args[])**

**{**

**String str="a";**

**String pat="[abc]";**

**Pattern pattern=Pattern.compile(pat);**

**Matcher matcher=pattern.matcher(str);**

**Boolean patternFoundFlag=matcher.matches();**

**System.out.println("The pattern is "+((patternFoundFlag==true)?"":"not ")+"found in the flag");**

**}**

**}**

Now, the pattern [abc] means it can either matches a or b or c.

Now, if str="a" and pattern="[abc]" then one character matching will be checked. and since, the pattern specifies that if the first and only character is one of a,b, or c a match will be found.

However, if str="ab" and pattern="[abc]" a match cannot be found. Since, pattern will expect that the string will have only one character.

**RegexExample7.java**

**import java.util.regex.Pattern;**

**import java.util.regex.Matcher;**

**class MatcherExample7**

**{**

**public static void main(String args[])**

**{**

**String text="fsdsgbavafaaaaa";**

**String pat="[^abc]\*";**

**Pattern pattern=Pattern.compile(pat);**

**Matcher matcher=pattern.matcher(text);**

**boolean patternFoundFlag=matcher.matches();**

**System.out.println("The pattern is "+((patternFoundFlag==true)?"":"not ")+"found in the text");**

**}**

**}**

Now, this is an example which contains the use of negation operator.

[^abc] means neither a nor b nor c. Now, the pattern is defined as [^abc]\*. Now, The result will be “The pattern is not found in the text”. Because, the string text contains a.

**RegexExample8.java**

import java.util.regex.Pattern;

import java.util.regex.Matcher;

class MatcherExample8

{

public static void main(String args[])

{

String text="ssbfsdgdaytefqrhwerrsndkeghgdsqqwrtyhvmdehj";

String patt="[a-z&&[^bx]]\*";

//this pattern tells it will match a string containing only small characters except b and c

//So, if the string contains any of b and c, pattern will not match

Pattern pattern=Pattern.compile(patt);

Matcher matcher=pattern.matcher(text);

boolean patternfoundflag=matcher.matches();

System.out.println("The pattern is "+((patternfoundflag==true)?"":"not ")+"found in the flag");

}

}

Now, if the text is ssfsdgdaytefqrhwerrsndkeghgdsqqwrtyhvmdehj

and pattern is [a-z&&[^bc]]\*

Now, as you can notice, there is no b or c in the text

the patt will match the text

However, if the text is: ssbfsdgdaytefqrhwerrsndkeghgdsqqwrtyhvmdehj i.e. if a single b is introduced to the text

the string pattern will not match.

**RegexExample9.java**

**import java.util.regex.Pattern;**

**import java.util.regex.Matcher;**

**class MatcherExample9**

**{**

**public static void main(String args[])**

**{**

**String text="ssbfsdgdaytefqrhwerrsndkeghgdsqqwrtyhvmdehj";**

**String patt="[a-z&&[^b-d]]\*";**

**//this pattern tells it will match a string containing only small characters except b and c and d**

**//So, if the string contains any of b and c and d, pattern will not match**

**Pattern pattern=Pattern.compile(patt);**

**Matcher matcher=pattern.matcher(text);**

**boolean patternfoundflag=matcher.matches();**

**System.out.println("The pattern is "+((patternfoundflag==true)?"":"not ")+"found in the flag");**

**}**

**}**

Now, as it can be noticed that the pattern is "[a-z&&[^b-d]]\*". The pattern tells it will match a string containing only small characters except b and c and d. So, the text won’t match with the pattern.

**RegexExample10.java**

**import java.util.regex.Pattern;**

**import java.util.regex.Matcher;**

**class Matcher**

**{**

**public static void main(String args[])**

**{**

**//Now, let's explore the union property in regex**

**String text="adssdhgjkjgfgxyz";**

**String pat="[a-s[x-z]]\*";**

**//Now, pattern says if a string consists characters from a-s or x-z**

**Pattern pattern=Pattern.compile(pat);**

**Matcher matcher=pattern.matcher(text);**

**boolean patternfoundflag=matcher.matches();**

**System.out.println("The pattern is "+((patternfoundflag==true)?"":"not ")+"found in the text");**

**String pat2="[a-sx-z]\*";**

**Pattern pattern2=Pattern.compile(pat2);**

**Matcher matcher2=pattern2.matcher(text);**

**boolean patternfoundflag2=matcher2.matches();**

**System.out.println("The pattern is "+((patternfoundflag2==true)?"":"not ")+"found in the text");**

**}**

**}**

Now, the pattern shows the union property in regex. Now, “[a-s[x-z]]\*” is the pattern defined. It means if a string consists of a-s or x-z, it will match with the pattern defined.

However, the pattern can also be defined as [a-sx-z].

**RegexExample11.java**

**import java.util.regex.Pattern;**

**import java.util.regex.Matcher;**

**class MatcherExample11**

**{**

**public static void main(String args[])**

**{**

**String text="asadaFA1eqefewase";**

**String patternToBeMatched="[a-zA-Z0-9]{7,18}";**

**Pattern pattern=Pattern.compile(patternToBeMatched);**

**Matcher matcher=pattern.matcher(text);**

**boolean patternfoundflag=matcher.matches();**

**System.out.println("The pattern is "+((patternfoundflag==true)?"":"not ")+"found in the text");**

**}**

**}**

Now, [a-zA-Z]{7,18} this pattern means the string will consist of upper case letters or lower case letter and the string will contain at least 7 letters and atmost 18 letters.

Now, if it is needed to define a pattern that contains at least 7 and atmost 18 alphanumeric characters **(like it is done here)** the pattern can be defined as [a-zA-Z0-9]{7,18}.

Or, It can be defined as [a-zA-Z\\d]{7,18}

Note, that [a-zA-Z\d]{7,18} won’t work. The program will give an error telling that \d is an unknown escape character.

**RegexExample12.java**

**import java.util.regex.Pattern;**

**import java.util.regex.Matcher;**

**class MatcherExample12**

**{**

**public static void main(String args[])**

**{**

**String text1="dsgdye5tsfsgterffaereefgafdfgaEEHRTEREAFGDFGQAA\r\fdsfg \n";**

**String pat1="(.|\\s)\*";**

**Pattern pattern=Pattern.compile(pat1);**

**Matcher matcher=pattern.matcher(text1);**

**boolean patternFoundFlag=matcher.matches();**

**System.out.println("The pattern is "+((patternFoundFlag==true)?"":"not ")+"found in text");**

**}**

}

Now, pat1=(.|\\s)\* this pattern will matches string which contains any character including \n,\r,\f etc.

Now, note that \s will be treated as unknown escape character, but \\s will work find for the same meaning.

**RegexExample13.java**

**import java.util.regex.Pattern;**

**import java.util.regex.Matcher;**

**class MatcherExample13**

**{**

**public static void main(String args[])throws IOException**

**{**

**String text="SDaft12asdaftt537";**

**String patt="[A-Za-z\\d]\*";**

**Pattern pattern=Pattern.compile(patt);**

**Matcher matcher=pattern.matcher(test);**

**boolean patternFoundFlag=matcher.matches();**

**System.out.println("The pattern is "+((patternFoundFlag==true)?"":"not ")+"found in the text");**

**}**

**}**

The result is The Pattern is found in the text.

Now, The pattern is defined as [A-Za-z\\d]\* which means only alphanumeric characters are allowed.

**RegexExample14.java**

**import java.util.regex.Pattern;**

**import java.util.regex.Matcher;**

**class MatcherExample14**

**{**

**public static void main(String args[])**

**{**

**String text="safafaacaAA1";**

**String patt="[A-Za-z\\D]\*";**

**Pattern pattern=Pattern.compile(patt);**

**Matcher matcher=pattern.matcher(text);**

**Boolean patternFoundFlag=matcher.matches();**

**System.out.println("The pattern is "+((patternFoundFlag==true)?"":"not ")+"found in the text");**

**}**

**}**

Now, this example shows the use of \D which is basically the negation of \d ([^\d] or [^0-9]).

Here, the pattern specified means alphabetic characters are allowed while numeric characters are not allowed.

However, this is an inappropriate example. A better example would be a pattern which matches all characters (including special characters and newline except numeric characters.

That can be defined as [(.|\\s)\\D]\*

Check the next example for that.

**RegexExample15.java**

**import java.util.regex.Pattern;**

**import java.util.regex.Matcher;**

**class MatcherExample15**

**{**

**public static void main(String args[])**

**{**

**String text="Afdsgdhdg1 !@/";**

**String pattern="[(.|\\s)\\D]\*";**

**Pattern patt=Pattern.compile(pattern);**

**Matcher matcher=patt.matcher(text);**

**boolean matchFoundFlag=matcher.matches();**

**System.out.println("The text is "+((matchFoundFlag==true)?"":"not ")+"matched with the pattern");**

**}**

**}**

Now, This will result as “The text is not matched with the pattern”.

However, if the text is changed to "Afdsgdhdg !@/" It will match the pattern.

**RegexExample16.java**

**import java.util.regex.Pattern;**

**import java.util.regex.Matcher;**

**class MatcherExample16**

**{**

**public static void main(String args[])**

**{**

**String text="I am Sayak-1";**

**String pat="(\\w|\\s)\*";**

**Pattern pattern=Pattern.compile(pat);**

**Matcher matcher=pattern.matcher(text);**

**boolean patternfoundflag=matcher.matches();**

**System.out.println("the pattern is "+((patternfoundflag==true)?"":"not ")+"found in the text");**

**}**

**}**

Now, [\w](\\\\w) is word character. It matches [A-Za-z0-9]

So, "(\\w|\\s)\*" the pattern matches with any character except special characters.

That’s why, I am Sayak1 matches the pattern. While, I am Sayak-1 does not match the pattern.

Now, the next example is a complex example:

**RegexExample17.java**

It validates a dotted decimal format IP.

**import java.util.regex.Pattern;**

**import java.util.regex.Matcher;**

**class MatcherExample17**

**{**

**public static void main(String args[])**

**{**

**String ipAddress="192.168.142.256";**

**String ipAddressPattern="(\\d{3}).(\\d{3}).(\\d{3}).(\\d{3})";**

**Pattern pattern=Pattern.compile(ipAddressPattern);**

**Matcher matcher=pattern.matcher(ipAddress);**

**if(matcher.matches())**

**{**

**int firstOctet=Integer.parseInt(matcher.group(1));**

**int secondOctet=Integer.parseInt(matcher.group(2));**

**int thirdOctet=Integer.parseInt(matcher.group(3));**

**int fourthOctet=Integer.parseInt(matcher.group(4));**

**System.out.println("First octet: "+firstOctet+" Second octet: "+secondOctet+" Third octet: "+thirdOctet+" Fourth Octet: "+fourthOctet);**

**if((firstOctet>=0&&firstOctet<=255)&&(secondOctet>=0&&secondOctet<=255)&&(thirdOctet>=0&&thirdOctet<=255)&&**

**(fourthOctet>=0&&fourthOctet<=255))**

**{**

**System.out.println("The ipaddress is a valid address");**

**}**

**else**

**{**

**System.out.println("The ipaddress is not a valid address");**

**}**

**}**

**else**

**{**

**System.out.println("The ipaddress is not a valid address");**

**}**

**}**

**}**

**RegexExample18.java**

**import java.util.regex.Pattern;**

**import java.util.regex.Matcher;**

**class MatcherExample18**

**{**

**public static void main(String args[])**

**{**

**String REGEX = "(^cat|cat$)";**

**String INPUT = "cat cat cat cattie cat";**

**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());**

**}**

**}**

**}**

Now, the pattern defines that if in the arbitrary character sequence, there's a cat phrase at the beginning of the line or there's a cat phrase at the end of the line a match will be found.

As a result you will see, two matches are found only in the given text.

Match number 1

start(): 0

end(): 3

Match number 2

start(): 19

end(): 22

**RegexExample19.java**

import java.util.regex.Pattern;

import java.util.regex.Matcher;

class RegexExample19

{

public static void main(String args[])

{

String text="[name]";

String patt1="[name]";

//the arbitrary character sequence

String patt2="\\Q[name]\\E";

Pattern pattern1=Pattern.compile(patt1);

Pattern pattern2=Pattern.compile(patt2);

Matcher matcher1=pattern1.matcher(text);

Matcher matcher2=pattern2.matcher(text);

boolean patternMatchesFlag=matcher1.matches();

System.out.println("Text is "+((patternMatchesFlag==true)?"":"not ")+"matched against patt1");

patternMatchesFlag=matcher2.matches();

System.out.println("Text is "+((patternMatchesFlag==true)?"":"not ")+"matched against pat2");

}

}

**RegexExample19.java**

import java.util.regex.Pattern;

import java.util.regex.Matcher;

class MatcherExample19

{

public static void main(String args[])

{

String text="[name]";

String patt1="[name]";

//the arbitrary character sequence

String patt2="\\Q[name]\\E";

Pattern pattern1=Pattern.compile(patt1);

Pattern pattern2=Pattern.compile(patt2);

Matcher matcher1=pattern1.matcher(text);

Matcher matcher2=pattern2.matcher(text);

boolean patternMatchesFlag=matcher1.matches();

System.out.println("Text is "+((patternMatchesFlag==true)?"":"not ")+"matched against patt1");

patternMatchesFlag=matcher2.matches();

System.out.println("Text is "+((patternMatchesFlag==true)?"":"not ")+"matched against pat2");

}

}

Now, the result is:

**Text is not matched against patt1**

**Text is matched against pat2**

**What’s the importance of \Q and \E?**

Java regex engine blocks special interpretation of all meta-characters between \Q and \E. For example, [name] matches a single character ('n', 'a', 'm', or 'e'), while \Q[name]\E matches six characters - '[', 'n', 'a', 'm', 'e', and '].

**RegexExample20.java**

import java.util.regex.Pattern;

import java.util.regex.Matcher;

class MatcherExample20

{

public static void main(String args[])

{

String regex="bar";

String charSeq="barrrrrrrrrr";

Pattern pattern=Pattern.compile(regex);

Matcher matcher=pattern.matcher(charSeq);

System.out.println("The charseq is "+((matcher.matches()==true)?"":"not ")+"matched with the regex");

System.out.println("The charseq is "+((matcher.lookingAt()==true)?"":"not ")+"partially matched with the regex");

}

}

This program will show the difference between lookingAt() and matches().

lookingAt() will found the pattern in the arbitrary character sequence, so, even if a partial portion of the arbitrary character sequence is matched with the regex, lookingAt will return true

Whereas, matches() will check if the whole arbitrary character sequence matches with the pattern or follows the pattern.

**RegexExample21.java**

import java.util.regex.Pattern;

import java.util.regex.Matcher;

class MatcherExample21

{

public static void main(String args[])

{

String toFind="dog";

String input="The dog says mew. All dogs say mew";

String toReplace="cat";

Pattern pattern=Pattern.compile(toFind);

Matcher matcher=pattern.matcher(input);

String output=matcher.replaceAll(toReplace);

System.out.println(output);

}

}

This is an example of replacing all occurrences of one pattern with another.

**RegexExample22.java**

**import java.util.regex.Pattern;**

**import java.util.regex.Matcher;**

**class MatcherExample22**

**{**

**public static void main(String args[])**

**{**

**String toFind="dog";**

**String input="The dog says mew. All dogs say mew";**

**String toReplace="cat";**

**Pattern pattern=Pattern.compile(toFind);**

**Matcher matcher=pattern.matcher(input);**

**String output=matcher.replaceFirst(toReplace);**

**System.out.println(output);**

**}**

**}**

This is an example of replacing the first occurrence of one pattern with another.

**RegexExample23.java**

**import java.util.regex.Matcher;**

**import java.util.regex.Pattern;**

**public class MatcherExample23**

**{**

**private static String REGEX = "a\*b";**

**private static String INPUT = "aabfooaabfooabfoob";**

**private static String REPLACE = "-";**

**public static void main(String[] args) {**

**Pattern p = Pattern.compile(REGEX);**

**// get a matcher object**

**Matcher m = p.matcher(INPUT);**

**StringBuffer sb = new StringBuffer();**

**while(m.find()) {**

**m.appendReplacement(sb, REPLACE);**

**}**

**m.appendTail(sb);**

**System.out.println(sb.toString());**

**}**

**}**

Now, This will produce the following result: -foo-foo-foo-

From the documentation:

public [Matcher](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html" \o "class in java.util.regex) appendReplacement([StringBuffer](https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuffer.html" \o "class in java.lang) sb,

[String](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html" \o "class in java.lang) replacement)

Implements a non-terminal append-and-replace step.

This method performs the following actions:

It reads characters from the input sequence, starting at the append position, and appends them to the given string buffer. It stops after reading the last character preceding the previous match, that is, the character at index [start()](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html" \l "start()) - 1.

It appends the given replacement string to the string buffer.

It sets the append position of this matcher to the index of the last character matched, plus one, that is, to [end()](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html" \l "end()).

The replacement string may contain references to subsequences captured during the previous match: Each occurrence of ${*name*} or $*g* will be replaced by the result of evaluating the corresponding [group(name)](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html" \l "group(java.lang.String)) or [group(g)](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html" \l "group(int))respectively. For $*g*, the first number after the $ is always treated as part of the group reference. Subsequent numbers are incorporated into g if they would form a legal group reference. Only the numerals '0' through '9' are considered as potential components of the group reference. If the second group matched the string "foo", for example, then passing the replacement string "$2bar" would cause "foobar" to be appended to the string buffer. A dollar sign ($) may be included as a literal in the replacement string by preceding it with a backslash (\$).

Note that backslashes (\) and dollar signs ($) in the replacement string may cause the results to be different than if it were being treated as a literal replacement string. Dollar signs may be treated as references to captured subsequences as described above, and backslashes are used to escape literal characters in the replacement string.

This method is intended to be used in a loop together with the [appendTail](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html" \l "appendTail(java.lang.StringBuffer)) and [find](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html" \l "find()) methods. The following code, for example, writes one dog two dogs in the yard to the standard-output stream:

Pattern p = Pattern.compile("cat");

Matcher m = p.matcher("one cat two cats in the yard");

StringBuffer sb = new StringBuffer();

while (m.find()) {

m.appendReplacement(sb, "dog");

}

m.appendTail(sb);

System.out.println(sb.toString());

****Parameters:****

sb - The target string buffer

replacement - The replacement string

**appendTail**:

Implements a terminal append-and-replace step.

This method reads characters from the input sequence, starting at the append position, and appends them to the given string buffer. It is intended to be invoked after one or more invocations of the [appendReplacement](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html" \l "appendReplacement(java.lang.StringBuffer, java.lang.String)) method in order to copy the remainder of the input sequence.

The explanation of basic examples end here.