**Regular expression:**

A regular expression is a sequence of characters that define a search pattern. Usually such pattern are used by “string searching” algorithms for “find” or “find and replace” operations on strings or for input validation.

.

**Regular expression:**

Email

Password Validations

Ms Word Find and Replace

See regex example

**Import java.util.regex**.\*;

Pattern Class //. Used for defining patterns

Matcher Class //. Used to perform match operations on text using patterns

These are two main classes

1. **import** java.util.regex.\*;
2. **public** **class** RegexExample1{
3. **public** **static** **void** main(String args[]){
4. //1st way
5. Pattern p = Pattern.compile(".s");//. represents single character
6. Matcher m = p.matcher("as");
7. **boolean** b = m.matches();
9. //2nd way
10. **boolean** b2=Pattern.compile(".s").matcher("as").matches();
12. //3rd way
13. **boolean** b3 = Pattern.matches(".s", "as");
15. System.out.println(b+" "+b2+" "+b3);
16. }}
17. **import** java.util.regex.\*;
18. **class** RegexExample2{
19. **public** **static** **void** main(String args[]){
20. System.out.println(Pattern.matches(".s", "as"));//true (2nd char is s)
21. System.out.println(Pattern.matches(".s", "mk"));//false (2nd char is not s)
22. System.out.println(Pattern.matches(".s", "mst"));//false (has more than 2 char)
23. System.out.println(Pattern.matches(".s", "amms"));//false (has more than 2 char)
24. System.out.println(Pattern.matches("..s", "mas"));//true (3rd char is s)
25. }}

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) |

1. **import** java.util.regex.\*;
2. **class** RegexExample3{
3. **public** **static** **void** main(String args[]){
4. System.out.println(Pattern.matches("[amn]", "abcd"));//false (not a or m or n)
5. System.out.println(Pattern.matches("[amn]", "a"));//true (among a or m or n)
6. System.out.println(Pattern.matches("[amn]", "ammmna"));//false (m and a comes more than once)
7. }}

Regex Quantifiers

The quantifiers specify the number of occurrences of a character.

|  |  |
| --- | --- |
| **Regex** | **Description** |
| X? | X occurs once or not at all (0 times or 1 time) |
| 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

1. **import** java.util.regex.\*;
2. **class** RegexExample4{
3. **public** **static** **void** main(String args[]){
4. System.out.println("? quantifier ....");
5. System.out.println(Pattern.matches("[amn]?", "a"));//true (a or m or n comes one time)
6. System.out.println(Pattern.matches("[amn]?", "aaa"));//false (a comes more than one time)
7. System.out.println(Pattern.matches("[amn]?", "aammmnn"));//false (a m and n comes more than one time)
8. System.out.println(Pattern.matches("[amn]?", "aazzta"));//false (a comes more than one time)
9. System.out.println(Pattern.matches("[amn]?", "am"));//false (a or m or n must come one time)
11. System.out.println("+ quantifier ....");
12. System.out.println(Pattern.matches("[amn]+", "a"));//true (a or m or n once or more times)
13. System.out.println(Pattern.matches("[amn]+", "aaa"));//true (a comes more than one time)
14. System.out.println(Pattern.matches("[amn]+", "aammmnn"));//true (a or m or n comes more than once)
15. System.out.println(Pattern.matches("[amn]+", "aazzta"));//false (z and t are not matching pattern)
17. System.out.println("\* quantifier ....");
18. System.out.println(Pattern.matches("[amn]\*", "ammmna"));//true (a or m or n may come zero or more times)
20. }}

Regex Metacharacters

The regular expression metacharacters work as shortcodes.

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

## Regular Expression Metacharacters Example

1. **import** java.util.regex.\*;
2. **class** RegexExample5{
3. **public** **static** **void** main(String args[]){
4. System.out.println("metacharacters d....");\\d means digit
6. System.out.println(Pattern.matches("\\d", "abc"));//false (non-digit)
7. System.out.println(Pattern.matches("\\d", "1"));//true (digit and comes once)
8. System.out.println(Pattern.matches("\\d", "4443"));//false (digit but comes more than once)
9. System.out.println(Pattern.matches("\\d", "323abc"));//false (digit and char)
11. System.out.println("metacharacters D....");\\D means non-digit
13. System.out.println(Pattern.matches("\\D", "abc"));//false (non-digit but comes more than once)
14. System.out.println(Pattern.matches("\\D", "1"));//false (digit)
15. System.out.println(Pattern.matches("\\D", "4443"));//false (digit)
16. System.out.println(Pattern.matches("\\D", "323abc"));//false (digit and char)
17. System.out.println(Pattern.matches("\\D", "m"));//true (non-digit and comes once)
19. System.out.println("metacharacters D with quantifier....");
20. System.out.println(Pattern.matches("\\D\*", "mak"));//true (non-digit and may come 0 or more times)
22. }}

String regex = "\\s";

String input = "Hello how are you";

Pattern p = Pattern.compile(regex);

Matcher m = p.matcher(input);

int count = 0;

while(m.find()) {

count++;

}

System.out.println("Number of matches: "+count);

1. /\*Create a regular expression that accepts alphanumeric characters only.
2. Its length must be six characters long only.\*/
4. **import** java.util.regex.\*;
5. **class** RegexExample6{
6. **public** **static** **void** main(String args[]){
7. System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun32"));//true
8. System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "kkvarun32"));//false (more than 6 char)
9. System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "JA2Uk2"));//true
10. System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun$2"));//false ($ is not matched)
11. }}
12. /\*Create a regular expression that accepts 10 digit numeric characters
13. starting with 7, 8 or 9 only.\*/
15. **import** java.util.regex.\*;
16. **class** RegexExample7{
17. **public** **static** **void** main(String args[]){
18. System.out.println("by character classes and quantifiers ...");
19. System.out.println(Pattern.matches("[789]{1}[0-9]{9}", "9953038949"));//true
20. System.out.println(Pattern.matches("[789][0-9]{9}", "9953038949"));//true
22. System.out.println(Pattern.matches("[789][0-9]{9}", "99530389490"));//false (11 characters)
23. System.out.println(Pattern.matches("[789][0-9]{9}", "6953038949"));//false (starts from 6)
24. System.out.println(Pattern.matches("[789][0-9]{9}", "8853038949"));//true
26. System.out.println("by metacharacters ...");
27. System.out.println(Pattern.matches("[789]{1}\\d{9}", "8853038949"));//true
28. System.out.println(Pattern.matches("[789]{1}\\d{9}", "3853038949"));//false (starts from 3)
30. }}

/\*Matcher class important methods

Boolean find():

It attept to find next match and return true if available otherwise it returns false.

Int start():

It return the starting index of the match.

Int end():

Return the index after last character matched.

String group():

Return the matched string

Main body

int count=0;

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

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

while(m.find())

{

count++;

System.out.println(m.start()+"------"+m.end()+"------"+m.group());

}

System.out.println("the number of occurance:"+count);

//take input from user

Scanner regxinput=new Scanner(System.in);

System.out.println("enter regular expression");

String regexexpre=regxinput.nextLine();

System.out.println("enter string");

String regularstring=regxinput.nextLine();

Boolean result= Pattern.matches(regexexpre, regularstring);

if(result)

{

System.out.println("string matched");

}

else

{

System.out.println("not matched");

}

# **Java StringBuffer class**

Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

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

private static String inputstring="aabfooaabfoobfoob";

private static String replace="-";

main body:

Pattern p=Pattern.compile(regexexp);

Matcher m=p.matcher(inputstring);

StringBuffer sb=new StringBuffer();

while(m.find())

{

m.appendReplacement(sb,replace);

}

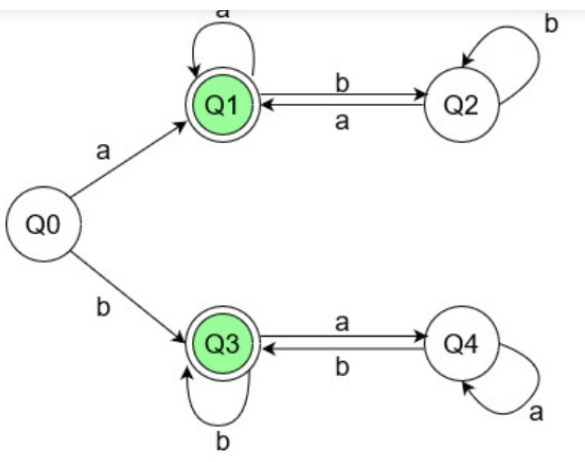
m.appendTail(sb);

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

Java static method

If you apply static keyword with any method, it is known as static method.

A static method can be invoked without the need for creating an instance of a class.



static void q1(String s, int i)

{

// Condition to check end of string

if (i == s.length())

{

System.out.println("Yes");

return;

}

// State transitions

// 'a' takes to q1, and

// 'b' takes to q2

if (s.charAt(i) == 'a')

q1(s, i + 1);

else

q2(s, i + 1);

}

// Function for the state Q2

static void q2(String s, int i)

{

// Condition to check end of string

if (i == s.length())

{

System.out.println("No");

return;

}

// State transitions

// 'a' takes to q1, and

// 'b' takes to q2

if (s.charAt(i) == 'a')

q1(s, i + 1);

else

q2(s, i + 1);

}

// Function for the state Q3

static void q3(String s, int i)

{

// Condition to check end of string

if (i == s.length())

{

System.out.println("Yes");

return;

}

// State transitions

// 'a' takes to q4, and

// 'b' takes to q3

if (s.charAt(i) == 'a')

q4(s, i + 1);

else

q3(s, i + 1);

}

// Function for the state Q4

static void q4(String s, int i)

{

// Condition to check end of string

if (i == s.length())

{

System.out.println("No");

return;

}

// State transitions

// 'a' takes to q4, and

// 'b' takes to q3

if (s.charAt(i) == 'a')

q4(s, i + 1);

else

q3(s, i + 1);

}

// Function for the state Q0

static void q0(String s, int i)

{

// Condition to check end of string

if (i == s.length())

{

System.out.println("No");

return;

}

// State transitions

// 'a' takes to q1, and

// 'b' takes to q3

if (s.charAt(i) == 'a')

q1(s, i + 1);

else

q3(s, i + 1);

}

public static void main(String[] args) {

String s = "aba";

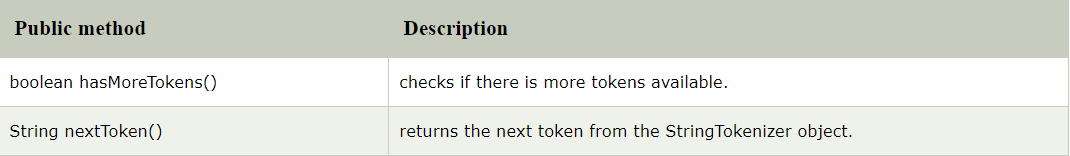
// Since q0 is the starting state

// Send the string to q0

q0(s, 0);

# Tokanization

Using tokanizer class // use for single as well as combination tokanization



String string\_to\_tok=string\_to\_tokenize.getText();

String tokenizer\_str=tokenizer\_string.getText();

String res="";

StringTokenizer token\_result = new StringTokenizer(string\_to\_tok, tokenizer\_str);

while (token\_result.hasMoreTokens()) {

res += token\_result.nextToken() + "\n";

}

System.out.println(res);

result\_txt.setText(res);

//

# using split method

String input\_string =input\_str.getText();

String regex\_str=regex.getText();

String [] result\_array=input\_string.split(regex\_str);

result\_box.setText(Arrays.toString(result\_array));

String str1 = "hexlloxyzhixyzbye";

String[] arr1 = str1.split("xyz"); // split mostly with xyz not single charater

for (String s2 : arr1)

{

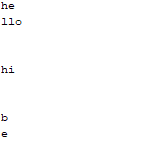
System.out.println(s2);

}

Output



String[] arr1 = str1.split("[xyz]");



Token numbers

String s1="welcome to split world";

for(String w:s1.split("e",2)){

System.out.println(w);

}



// 0 will give you the complete string no tokenization is done

First and Follow

S -> A B

A -> a B

B -> s

Terminals(can not change lowercase) and non-Terminals(can change uppercase)

Find terminal and non terminal using split method

Before code we need to check these concepts

* Array list
* Hash Set
* Hash Map

Array List

The ArrayList class is a resizable [array](https://www.w3schools.com/java/java_arrays.asp), which can be found in the java.util package.

import java.util.ArrayList; // import the ArrayList class

ArrayList<String> cars = new ArrayList<String>();

cars.add("Volvo"); // add method

System.out.println(cars);

Hash Map

A HashMap however, store items in "**key**/**value**" pairs, and you can access them by an index of another type (e.g. a String).

It can store different types: String keys and Integer values, or the same type, like: String keys and String values:

import java.util.HashMap

Create a HashMap object called capitalCities that will store String keys and String values:

HashMap<String, String> capitalCities = new HashMap<String, String>();

// Add keys and values (Country, City)

capitalCities.put("England", "London");

capitalCities.put("Germany", "Berlin");

capitalCities.put("Norway", "Oslo");

capitalCities.put("USA", "Washington DC");

System.out.println(capitalCities);

Hash Set

A HashSet is a collection of items where every item is unique, and it is found in the java.util package:

**Note:** In the example above, even though BMW is added twice it only appears once in the set because every item in a set has to be unique.

// Create a HashSet object called numbers

HashSet<Integer> numbers = new HashSet<Integer>();

//access hashset data

**for-each** loop:

for (String i : numbers) {

System.out.println(i);

}

A list can store duplicate value but set can not.

Clone

1. **class** Student18 **implements** Cloneable{
2. **int** rollno;
3. String name;
5. Student18(**int** rollno,String name){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. }
10. **public** Object clone()**throws** CloneNotSupportedException{
11. **return** **super**.clone();
12. }
14. **public** **static** **void** main(String args[]){
15. **try**{
16. Student18 s1=**new** Student18(101,"amit");
18. Student18 s2=(Student18)s1.clone();
20. System.out.println(s1.rollno+" "+s1.name);
21. System.out.println(s2.rollno+" "+s2.name);
23. }**catch**(CloneNotSupportedException c){}
25. }
26. }

**@Override**

The**@Override** annotation is one of a default Java annotation and it can be introduced in **Java 1.5** Version. The **@Override** annotation indicates that the **child class method is over-writing its base class method**.

Example:

// A Simple Java program to demonstrate

// method overriding in java

// Base Class

class Parent {

void show()

{

System.out.println("Parent's show()");

}

}

// Inherited class

class Child extends Parent {

// This method overrides show() of Parent

@Override

void show()

{

System.out.println("Child's show()");

}

}

// Driver class

class Main {

public static void main(String[] args)

{

// If a Parent type reference refers

// to a Parent object, then Parent's

// show is called

Parent obj1 = new Parent();

obj1.show();

// If a Parent type reference refers

// to a Child object Child's show()

// is called. This is called RUN TIME

// POLYMORPHISM.

Parent obj2 = new Child();

obj2.show();

}

}

S -> A B c | g h i | j k l

A -> a|b|c

B -> b

D -> d

S -> A B C

A -> a | b | ε

B -> c | d | ε

C -> e | f | ε