Java Regex

The **Java Regex** or Regular Expression is an API to *define a pattern for searching or manipulating strings*.

It is widely used to define the constraint on strings such as password and email validation. After learning Java regex tutorial, you will be able to test your regular expressions by the Java Regex Tester Tool.

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

java.util.regex package

The Matcher and Pattern classes provide the facility of Java regular expression. The java.util.regex package provides following classes and interfaces for regular expressions.

1. MatchResult interface
2. Matcher class
3. Pattern class
4. PatternSyntaxException class

A regular expression is a sequence of characters that forms a search pattern. When you search for data in a text, you can use this search pattern to describe what you are searching for.

A regular expression can be a single character, or a more complicated pattern.

Regular expressions can be used to perform all types of **text search** and **text replace** operations.

## **Matcher class**

It implements the **MatchResult** interface. It is a regex engine which is 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 returns the instance of the Pattern. |
| 2 | Matcher matcher(CharSequence input) | creates a matcher that matches the given input with the 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.

1. **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);

}}

## **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) |

## **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 |

## **egex 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, 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**](https://www.javatpoint.com/opr/test.jsp?filename=RegexExample5)

## **Regular Expression Question 1**

1. /\*Create a regular expression that accepts alphanumeric characters only.

Its length must be six 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)

}}

# Java Lambda Expressions

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

Java lambda expression is treated as a function, so compiler does not create .class file.

## **Functional Interface**

Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface. Java provides an anotation @FunctionalInterface, which is used to declare an interface as functional interface.

## **Why use Lambda Expression**

1. To provide the implementation of Functional interface.
2. Less coding.

## **Java Lambda Expression Syntax**

1. (argument-list) -> {body}

## **Without Lambda Expression**

**interface** Drawable{

**public** **void** draw();

}

**public** **class** LambdaExpressionExample {

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

**int** width=10;

        //without lambda, Drawable implementation using anonymous class

        Drawable d=**new** Drawable(){

**public** **void** draw(){System.out.println("Drawing "+width);}

        };

        d.draw();

    }

}

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample)

Output:

Drawing 10

## **Java Lambda Expression Example**

Now, we are going to implement the above example with the help of Java lambda expression.

@FunctionalInterface  //It is optional

**interface** Drawable{

**public** **void** draw();

}

**public** **class** LambdaExpressionExample2 {

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

**int** width=10;

        //with lambda

        Drawable d2=()->{

            System.out.println("Drawing "+width);

        };

        d2.draw();

    }

}

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample2)

Output:

Drawing 10

## **Java Lambda Expression Example: Multiple Parameters**

**interface** Addable{

**int** add(**int** a,**int** b);

}

**public** **class** LambdaExpressionExample5{

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

        // Multiple parameters in lambda expression

        Addable ad1=(a,b)->(a+b);

        System.out.println(ad1.add(10,20));

        // Multiple parameters with data type in lambda expression

        Addable ad2=(**int** a,**int** b)->(a+b);

        System.out.println(ad2.add(100,200));

    }

}

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample5)

Output:

30

300

## **Java Lambda Expression Example: Foreach Loop**

**import** java.util.\*;

**public** **class** LambdaExpressionExample7{

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

        List<String> list=**new** ArrayList<String>();

        list.add("ankit");

        list.add("mayank");

        list.add("irfan");

        list.add("jai");

        list.forEach(

            (n)->System.out.println(n)

        );

    }

}

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample7)

Output:

ankit

mayank

irfan

jai

**Java lambda expression can be used in the collection framework. It provides efficient and concise way to iterate, filter and fetch data.**

# Java Functional Interfaces

An Interface that contains exactly one abstract method is known as functional interface.

# Java 8 Stream

Java provides a new additional package in Java 8 called java.util.stream. This package consists of classes, interfaces and enum to allows functional-style operations on the elements. You can use stream by importing java.util.stream package.

Stream provides following features:

* Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
* Stream is functional in nature. Operations performed on a stream does not modify it's source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
* Stream is lazy and evaluates code only when required.
* The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

You can use stream to filter, collect, print, and convert from one data structure to other etc. In the following examples, we have apply various operations with the help of stream.

### **Java Stream Example: Filtering Collection by using Stream**

Here, we are filtering data by using stream. You can see that code is optimized and maintained. Stream provides fast execution.

**import** java.util.\*;

**import** java.util.stream.Collectors;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** JavaStreamExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        List<Float> productPriceList2 =productsList.stream()

                                     .filter(p -> p.price > 30000)// filtering data

                                     .map(p->p.price)        // fetching price

                                     .collect(Collectors.toList()); // collecting as list

        System.out.println(productPriceList2);

    }

}

**Output:**

[90000.0]

### **Java Stream Iterating Example**

You can use stream to iterate any number of times. Stream provides predefined methods to deal with the logic you implement. In the following example, we are iterating, filtering and passed a limit to fix the iteration.

**import** java.util.stream.\*;

**public** **class** JavaStreamExample {

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

        Stream.iterate(1, element->element+1)

        .filter(element->element%5==0)

        .limit(5)

        .forEach(System.out::println);

    }

}

**Output:**

5

10

15

20

25

  // This is more compact approach for filtering data

        Float totalPrice = productsList.stream()

                    .map(product->product.price)

                    .reduce(0.0f,(sum, price)->sum+price);   // accumulating price

        System.out.println(totalPrice);

// count number of products based on the filter

**long** count = productsList.stream()

                    .filter(product->product.price<30000)

                    .count();

        System.out.println(count);

1. / Converting product List into Set
2. Set<Float> productPriceList =
3. productsList.stream()
4. .filter(product->product.price < 30000)   // filter product on the base of price
5. .map(product->product.price)
6. .collect(Collectors.toSet());   // collect it as Set(remove duplicate elements)
7. System.out.println(productPriceList);

   // Converting Product List into a Map

        Map<Integer,String> productPriceMap =

            productsList.stream()

                        .collect(Collectors.toMap(p->p.id, p->p.name));

        System.out.println(productPriceMap);

# Java forEach loop

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interface. It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach loop to iterate elements.

This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

## **forEach() Signature in Iterable Interface**

1. **default** **void** forEach(Consumer<**super** T>action)

### **Java 8 forEach() example 1**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** ForEachExample {

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

        List<String> gamesList = **new** ArrayList<String>();

        gamesList.add("Football");

        gamesList.add("Cricket");

        gamesList.add("Chess");

        gamesList.add("Hocky");

        System.out.println("------------Iterating by passing lambda expression--------------");

        gamesList.forEach(games -> System.out.println(games));

    }

}

Output:

------------Iterating by passing lambda expression--------------

Football

Cricket

Chess

Hocky