New Features

There are dozens of features added to Java 8, the most significant ones are mentioned below −

* **Lambda expression** − Adds functional processing capability to Java.
* **Method references** − Referencing functions by their names instead of invoking them directly. Using functions as parameter.
* **Default method** − Interface to have default method implementation.
* **New tools** − New compiler tools and utilities are added like ‘jdeps’ to figure out dependencies.
* **Stream API** − New stream API to facilitate pipeline processing.
* **Date Time API** − Improved date time API.
* **Optional** − Emphasis on best practices to handle null values properly.
* **Nashorn, JavaScript Engine** − A Java-based engine to execute JavaScript code.

Along with these new featuers, lots of feature enhancements are done under-the-hood, at both compiler and JVM level.

# Java 8 - Lambda Expressions

Lambda expressions are introduced in Java 8 and are touted to be the biggest feature of Java 8. Lambda expression facilitates functional programming, and simplifies the development a lot.

## Syntax

A lambda expression is characterized by the following syntax −

parameter -> expression body

Following are the important characteristics of a lambda expression −

* **Optional type declaration** − No need to declare the type of a parameter. The compiler can inference the same from the value of the parameter.
* **Optional parenthesis around parameter** − No need to declare a single parameter in parenthesis. For multiple parameters, parentheses are required.
* **Optional curly braces** − No need to use curly braces in expression body if the body contains a single statement.
* **Optional return keyword** − The compiler automatically returns the value if the body has a single expression to return the value. Curly braces are required to indicate that expression returns a value.

## Lambda Expressions Example

Create the following Java program using editor and save in some folder like C:\>JAVA.

### Java8Tester.java

public class Java8Tester {

public static void main(String args[]){

Java8Tester tester = new Java8Tester();

//with type declaration

MathOperation addition = (int a, int b) -> a + b;

//with out type declaration

MathOperation subtraction = (a, b) -> a - b;

//with return statement along with curly braces

MathOperation multiplication = (int a, int b) -> { return a \* b; };

//without return statement and without curly braces

MathOperation division = (int a, int b) -> a / b;

System.out.println("10 + 5 = " + tester.operate(10, 5, addition));

System.out.println("10 - 5 = " + tester.operate(10, 5, subtraction));

System.out.println("10 x 5 = " + tester.operate(10, 5, multiplication));

System.out.println("10 / 5 = " + tester.operate(10, 5, division));

//with parenthesis

GreetingService greetService1 = message ->

System.out.println("Hello " + message);

//without parenthesis

GreetingService greetService2 = (message) ->

System.out.println("Hello " + message);

greetService1.sayMessage("Mahesh");

greetService2.sayMessage("Suresh");

}

interface MathOperation {

int operation(int a, int b);

}

interface GreetingService {

void sayMessage(String message);

}

private int operate(int a, int b, MathOperation mathOperation){

return mathOperation.operation(a, b);

}

}

### Verify the Result

Compile the class using **javac** compiler as follows −

$javac Java8Tester.java

Now run the Java8Tester as follows −

$java Java8Tester

It should produce the following output −

10 + 5 = 15

10 - 5 = 5

10 x 5 = 50

10 / 5 = 2

Hello Mahesh

Hello Suresh

Following are the important points to be considered in the above example.

* Lambda expressions are used primarily to define inline implementation of a functional interface, i.e., an interface with a single method only. In the above example, we've used various types of lambda expressions to define the operation method of MathOperation interface. Then we have defined the implementation of sayMessage of GreetingService.
* Lambda expression eliminates the need of anonymous class and gives a very simple yet powerful functional programming capability to Java.

## Scope

Using lambda expression, you can refer to final variable or effectively final variable (which is assigned only once). Lambda expression throws a compilation error, if a variable is assigned a value the second time.

### Scope Example

Create the following Java program using editor and save in some folder like C:\>JAVA.

**Java8Tester.java**

public class Java8Tester {

final static String salutation = "Hello! ";

public static void main(String args[]){

GreetingService greetService1 = message ->

System.out.println(salutation + message);

greetService1.sayMessage("Mahesh");

}

interface GreetingService {

void sayMessage(String message);

}

}

### Verify the Result

Compile the class using **javac** compiler as follows −

$javac Java8Tester.java

Now run the Java8Tester as follows −

$java Java8Tester

It should produce the following output −

Hello! Mahesh

# Java 8 - Default Methods

Java 8 introduces a new concept of default method implementation in interfaces. This capability is added for backward compatibility so that old interfaces can be used to leverage the lambda expression capability of Java 8. For example, ‘List’ or ‘Collection’ interfaces do not have ‘forEach’ method declaration. Thus, adding such method will simply break the collection framework implementations. Java 8 introduces default method so that List/Collection interface can have a default implementation of forEach method, and the class implementing these interfaces need not implement the same.

## Syntax

public interface vehicle {

default void print(){

System.out.println("I am a vehicle!");

}

}

## Multiple Defaults

With default functions in interfaces, there is a possibility that a class is implementing two interfaces with same default methods. The following code explains how this ambiguity can be resolved.

public interface vehicle {

default void print(){

System.out.println("I am a vehicle!");

}

}

public interface fourWheeler {

default void print(){

System.out.println("I am a four wheeler!");

}

}

First solution is to create an own method that overrides the default implementation.

public class car implements vehicle, fourWheeler {

default void print(){

System.out.println("I am a four wheeler car vehicle!");

}

}

Second solution is to call the default method of the specified interface using super.

public class car implements vehicle, fourWheeler {

default void print(){

vehicle.super.print();

}

}

## Static Default Methods

An interface can also have static helper methods from Java 8 onwards.

public interface vehicle {

default void print(){

System.out.println("I am a vehicle!");

}

static void blowHorn(){

System.out.println("Blowing horn!!!");

}

}

## Default Method Example

Let's see an example to get more clarity on **default method**. Please write the following program in an code editor, understand and verify the results.

### Java8Tester.java

public class Java8Tester {

public static void main(String args[]){

Vehicle vehicle = new Car();

vehicle.print();

}

}

interface Vehicle {

default void print(){

System.out.println("I am a vehicle!");

}

static void blowHorn(){

System.out.println("Blowing horn!!!");

}

}

interface FourWheeler {

default void print(){

System.out.println("I am a four wheeler!");

}

}

class Car implements Vehicle, FourWheeler {

public void print(){

Vehicle.super.print();

FourWheeler.super.print();

Vehicle.blowHorn();

System.out.println("I am a car!");

}

}

### Verify the Result

Compile the class using **javac** compiler as follows −

$javac Java8Tester.java

Now run the Java8Tester as follows −

$java Java8Tester

It should produce the following output −

I am a vehicle!

I am a four wheeler!

Blowing horn!!!

I am a car!

# Java 8 - Streams

Stream is a new abstract layer introduced in Java 8. Using stream, you can process data in a declarative way similar to SQL statements. For example, consider the following SQL statement −

SELECT max(salary), employee\_id, employee\_name FROM Employee

The above SQL expression automatically returns the maximum salaried employee's details, without doing any computation on the developer's end. Using collections framework in Java, a developer has to use loops and make repeated checks. Another concern is efficiency; as multi-core processors are available at ease, a Java developer has to write parallel code processing that can be pretty error-prone.

To resolve such issues, Java 8 introduced the concept of stream that lets the developer to process data declaratively and leverage multicore architecture without the need to write any specific code for it.

## What is Stream?

Stream represents a sequence of objects from a source, which supports aggregate operations. Following are the characteristics of a Stream −

* **Sequence of elements** − A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.
* **Source** − Stream takes Collections, Arrays, or I/O resources as input source.
* **Aggregate operations** − Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.
* **Pipelining** − Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.
* **Automatic iterations** − Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

## Generating Streams

With Java 8, Collection interface has two methods to generate a Stream −

* **stream()** − Returns a sequential stream considering collection as its source.
* **parallelStream()** − Returns a parallel Stream considering collection as its source.

List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());

## forEach

Stream has provided a new method ‘forEach’ to iterate each element of the stream. The following code segment shows how to print 10 random numbers using forEach.

Random random = new Random();

random.ints().limit(10).forEach(System.out::println);

## map

The ‘map’ method is used to map each element to its corresponding result. The following code segment prints unique squares of numbers using map.

List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

//get list of unique squares

List<Integer> squaresList = numbers.stream().map( i -> i\*i).distinct().collect(Collectors.toList());

## filter

The ‘filter’ method is used to eliminate elements based on a criteria. The following code segment prints a count of empty strings using filter.

List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string

int count = strings.stream().filter(string -> string.isEmpty()).count();

## limit

The ‘limit’ method is used to reduce the size of the stream. The following code segment shows how to print 10 random numbers using limit.

Random random = new Random();

random.ints().limit(10).forEach(System.out::println);

## sorted

The ‘sorted’ method is used to sort the stream. The following code segment shows how to print 10 random numbers in a sorted order.

Random random = new Random();

random.ints().limit(10).sorted().forEach(System.out::println);

## Parallel Processing

parallelStream is the alternative of stream for parallel processing. Take a look at the following code segment that prints a count of empty strings using parallelStream.

List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string

int count = strings.parallelStream().filter(string -> string.isEmpty()).count();

It is very easy to switch between sequential and parallel streams.

## Collectors

Collectors are used to combine the result of processing on the elements of a stream. Collectors can be used to return a list or a string.

List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());

System.out.println("Filtered List: " + filtered);

String mergedString = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.joining(", "));

System.out.println("Merged String: " + mergedString);

## Statistics

With Java 8, statistics collectors are introduced to calculate all statistics when stream processing is being done.

List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

IntSummaryStatistics stats = integers.stream().mapToInt((x) -> x).summaryStatistics();

System.out.println("Highest number in List : " + stats.getMax());

System.out.println("Lowest number in List : " + stats.getMin());

System.out.println("Sum of all numbers : " + stats.getSum());

System.out.println("Average of all numbers : " + stats.getAverage());

## Stream Example

Create the following Java program using any editor of your choice in, say, C:\> JAVA.

### Java8Tester.java

import java.util.ArrayList;

import java.util.Arrays;

import java.util.IntSummaryStatistics;

import java.util.List;

import java.util.Random;

import java.util.stream.Collectors;

import java.util.Map;

public class Java8Tester {

public static void main(String args[]){

System.out.println("Using Java 7: ");

// Count empty strings

List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

System.out.println("List: " +strings);

long count = getCountEmptyStringUsingJava7(strings);

System.out.println("Empty Strings: " + count);

count = getCountLength3UsingJava7(strings);

System.out.println("Strings of length 3: " + count);

//Eliminate empty string

List<String> filtered = deleteEmptyStringsUsingJava7(strings);

System.out.println("Filtered List: " + filtered);

//Eliminate empty string and join using comma.

String mergedString = getMergedStringUsingJava7(strings,", ");

System.out.println("Merged String: " + mergedString);

List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

//get list of square of distinct numbers

List<Integer> squaresList = getSquares(numbers);

System.out.println("Squares List: " + squaresList);

List<Integer> integers = Arrays.asList(1,2,13,4,15,6,17,8,19);

System.out.println("List: " +integers);

System.out.println("Highest number in List : " + getMax(integers));

System.out.println("Lowest number in List : " + getMin(integers));

System.out.println("Sum of all numbers : " + getSum(integers));

System.out.println("Average of all numbers : " + getAverage(integers));

System.out.println("Random Numbers: ");

//print ten random numbers

Random random = new Random();

for(int i=0; i < 10; i++){

System.out.println(random.nextInt());

}

System.out.println("Using Java 8: ");

System.out.println("List: " +strings);

count = strings.stream().filter(string->string.isEmpty()).count();

System.out.println("Empty Strings: " + count);

count = strings.stream().filter(string -> string.length() == 3).count();

System.out.println("Strings of length 3: " + count);

filtered = strings.stream().filter(string ->!string.isEmpty()).collect(Collectors.toList());

System.out.println("Filtered List: " + filtered);

mergedString = strings.stream().filter(string ->!string.isEmpty()).collect(Collectors.joining(", "));

System.out.println("Merged String: " + mergedString);

squaresList = numbers.stream().map( i ->i\*i).distinct().collect(Collectors.toList());

System.out.println("Squares List: " + squaresList);

System.out.println("List: " +integers);

IntSummaryStatistics stats = integers.stream().mapToInt((x) ->x).summaryStatistics();

System.out.println("Highest number in List : " + stats.getMax());

System.out.println("Lowest number in List : " + stats.getMin());

System.out.println("Sum of all numbers : " + stats.getSum());

System.out.println("Average of all numbers : " + stats.getAverage());

System.out.println("Random Numbers: ");

random.ints().limit(10).sorted().forEach(System.out::println);

//parallel processing

count = strings.parallelStream().filter(string -> string.isEmpty()).count();

System.out.println("Empty Strings: " + count);

}

private static int getCountEmptyStringUsingJava7(List<String> strings){

int count = 0;

for(String string: strings){

if(string.isEmpty()){

count++;

}

}

return count;

}

private static int getCountLength3UsingJava7(List<String> strings){

int count = 0;

for(String string: strings){

if(string.length() == 3){

count++;

}

}

return count;

}

private static List<String> deleteEmptyStringsUsingJava7(List<String> strings){

List<String> filteredList = new ArrayList<String>();

for(String string: strings){

if(!string.isEmpty()){

filteredList.add(string);

}

}

return filteredList;

}

private static String getMergedStringUsingJava7(List<String> strings, String separator){

StringBuilder stringBuilder = new StringBuilder();

for(String string: strings){

if(!string.isEmpty()){

stringBuilder.append(string);

stringBuilder.append(separator);

}

}

String mergedString = stringBuilder.toString();

return mergedString.substring(0, mergedString.length()-2);

}

private static List<Integer> getSquares(List<Integer> numbers){

List<Integer> squaresList = new ArrayList<Integer>();

for(Integer number: numbers){

Integer square = new Integer(number.intValue() \* number.intValue());

if(!squaresList.contains(square)){

squaresList.add(square);

}

}

return squaresList;

}

private static int getMax(List<Integer> numbers){

int max = numbers.get(0);

for(int i=1;i < numbers.size();i++){

Integer number = numbers.get(i);

if(number.intValue() > max){

max = number.intValue();

}

}

return max;

}

private static int getMin(List<Integer> numbers){

int min = numbers.get(0);

for(int i=1;i < numbers.size();i++){

Integer number = numbers.get(i);

if(number.intValue() < min){

min = number.intValue();

}

}

return min;

}

private static int getSum(List numbers){

int sum = (int)(numbers.get(0));

for(int i=1;i < numbers.size();i++){

sum += (int)numbers.get(i);

}

return sum;

}

private static int getAverage(List<Integer> numbers){

return getSum(numbers) / numbers.size();

}

}

### Verify the Result

Compile the class using **javac** compiler as follows −

$javac Java8Tester.java

Now run the Java8Testeras follows −

$java Java8Tester

It should produce the following result −

Using Java 7:

List: [abc, , bc, efg, abcd, , jkl]

Empty Strings: 2

Strings of length 3: 3

Filtered List: [abc, bc, efg, abcd, jkl]

Merged String: abc, bc, efg, abcd, jkl

Squares List: [9, 4, 49, 25]

List: [1, 2, 13, 4, 15, 6, 17, 8, 19]

Highest number in List : 19

Lowest number in List : 1

Sum of all numbers : 85

Average of all numbers : 9

Random Numbers:

-1279735475

903418352

-1133928044

-1571118911

628530462

18407523

-881538250

-718932165

270259229

421676854

Using Java 8:

List: [abc, , bc, efg, abcd, , jkl]

Empty Strings: 2

Strings of length 3: 3

Filtered List: [abc, bc, efg, abcd, jkl]

Merged String: abc, bc, efg, abcd, jkl

Squares List: [9, 4, 49, 25]

List: [1, 2, 13, 4, 15, 6, 17, 8, 19]

Highest number in List : 19

Lowest number in List : 1

Sum of all numbers : 85

Average of all numbers : 9.444444444444445

Random Numbers:

-1009474951

-551240647

-2484714

181614550

933444268

1227850416

1579250773

1627454872

1683033687

1798939493

Empty Strings: 2

# Java 8 - Nashorn JavaScript

With Java 8, Nashorn, a much improved javascript engine is introduced, to replace the existing Rhino. Nashorn provides 2 to 10 times better performance, as it directly compiles the code in memory and passes the bytecode to JVM. Nashorn uses **invokedynamics** feature, introduced in Java 7 to improve performance.

## jjs

For Nashorn engine, JAVA 8 introduces a new command line tool, **jjs,** to execute javascript codes at console.

### Interpreting js File

Create and save the file **sample.js** in c:\> JAVA folder.

### sample.js

print('Hello World!');

Open console and use the following command.

$jjs sample.js

It will produce the following output:

Hello World!

### jjs in Interactive Mode

Open the console and use the following command.

$jjs

jjs> print("Hello, World!")

Hello, World!

jjs> quit()

>>

### Pass Arguments

Open the console and use the following command.

$jjs -- a b c

jjs> print('letters: ' +arguments.join(", "))

letters: a, b, c

jjs>

## Calling JavaScript from Java

Using ScriptEngineManager, JavaScript code can be called and interpreted in Java.

### Example

Create the following Java program using any editor of your choice in, say, C:\> JAVA.

### Java8Tester.java

import javax.script.ScriptEngineManager;

import javax.script.ScriptEngine;

import javax.script.ScriptException;

public class Java8Tester {

public static void main(String args[]){

ScriptEngineManager scriptEngineManager = new ScriptEngineManager();

ScriptEngine nashorn = scriptEngineManager.getEngineByName("nashorn");

String name = "Mahesh";

Integer result = null;

try {

nashorn.eval("print('" + name + "')");

result = (Integer) nashorn.eval("10 + 2");

}catch(ScriptException e){

System.out.println("Error executing script: "+ e.getMessage());

}

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

}

}

### Verify the Result

Compile the class using **javac** compiler as follows −

$javac Java8Tester.java

Now run the Java8Tester as follows −

$java Java8Tester

It should produce the following result −

Mahesh

12

## Calling Java from JavaScript

The following example explains how to import and use Java classes in java script −

### sample.js

var BigDecimal = Java.type('java.math.BigDecimal');

function calculate(amount, percentage) {

var result = new BigDecimal(amount).multiply(

new BigDecimal(percentage)).divide(new BigDecimal("100"), 2, BigDecimal.ROUND\_HALF\_EVEN);

return result.toPlainString();

}

var result = calculate(568000000000000000023,13.9);

print(result);

Open the console and use the following command.

$jjs sample.js

It should produce the following output −

78952000000000000003.20

# Java 8 - Base64

With Java 8, Base64 has finally got its due. Java 8 now has inbuilt encoder and decoder for Base64 encoding. In Java 8, we can use three types of Base64 encoding −

* **Simple** − Output is mapped to a set of characters lying in **A-Za-z0-9+/**. The encoder does not add any line feed in output, and the decoder rejects any character other than A-Za-z0-9+/.
* **URL** − Output is mapped to set of characters lying in **A-Za-z0-9+\_**. Output is URL and filename safe.
* **MIME** − Output is mapped to MIME friendly format. Output is represented in lines of no more than 76 characters each, and uses a carriage return '\r' followed by a linefeed '\n' as the line separator. No line separator is present to the end of the encoded output.

## Nested Classes

|  |  |
| --- | --- |
| **S. No.** | **Nested class & Description** |
| 1 | **static class Base64.Decoder**  This class implements a decoder for decoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045. |
| 2 | **static class Base64.Encoder**  This class implements an encoder for encoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045. |

## Methods

|  |  |
| --- | --- |
| **S. No.** | **Method Name & Description** |
| 1 | **static Base64.Decoder getDecoder()**  Returns a Base64.Decoder that decodes using the Basic type base64 encoding scheme. |
| 2 | **static Base64.Encoder getEncoder()**  Returns a Base64.Encoder that encodes using the Basic type base64 encoding scheme. |
| 3 | **static Base64.Decoder getMimeDecoder()**  Returns a Base64.Decoder that decodes using the MIME type base64 decoding scheme. |
| 4 | **static Base64.Encoder getMimeEncoder()**  Returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme. |
| 5 | **static Base64.Encoder getMimeEncoder(int lineLength, byte[] lineSeparator)**  Returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme with specified line length and line separators. |
| 6 | **static Base64.Decoder getUrlDecoder()**  Returns a Base64.Decoder that decodes using the URL and Filename safe type base64 encoding scheme. |
| 7 | **static Base64.Encoder getUrlEncoder()**  Returns a Base64.Encoder that encodes using the URL and Filename safe type base64 encoding scheme. |

## Methods Inherited

Base64 class inherits few methods from the **java.lang.Object** class.

## Base64 Example

Let us see an example to see Base64 in practice −

### Java8Tester.java

import java.util.Base64;

import java.util.UUID;

import java.io.UnsupportedEncodingException;

public class HelloWorld {

public static void main(String args[]){

try {

// Encode using basic encoder

String base64encodedString = Base64.getEncoder().encodeToString("TutorialsPoint?java8".getBytes("utf-8"));

System.out.println("Base64 Encoded String (Basic) :" + base64encodedString);

// Decode

byte[] base64decodedBytes = Base64.getDecoder().decode(base64encodedString);

System.out.println("Original String: " + new String(base64decodedBytes, "utf-8"));

base64encodedString = Base64.getUrlEncoder().encodeToString("TutorialsPoint?java8".getBytes("utf-8"));

System.out.println("Base64 Encoded String (URL) :" + base64encodedString);

StringBuilder stringBuilder = new StringBuilder();

for (int i = 0; i < 10; ++i) {

stringBuilder.append(UUID.randomUUID().toString());

}

byte[] mimeBytes = stringBuilder.toString().getBytes("utf-8");

String mimeEncodedString = Base64.getMimeEncoder().encodeToString(mimeBytes);

System.out.println("Base64 Encoded String (MIME) :" + mimeEncodedString);

}catch(UnsupportedEncodingException e){

System.out.println("Error :" + e.getMessage());

}

}

}

### Verify the Result

Compile the class using **javac** compiler as follows −

$javac Java8Tester.java

Now run the Java8Tester as follows −

$java Java8Tester

It should produce the following output −

Base64 Encoded String (Basic) :VHV0b3JpYWxzUG9pbnQ/amF2YTg=

Original String: TutorialsPoint?java8

Base64 Encoded String (URL) :VHV0b3JpYWxzUG9pbnQ\_amF2YTg=

Base64 Encoded String (MIME) :ZWJjY2YzZWUtYmUwZC00Yjg1LTlkYjUtNWUyMzBlNWI

4ZGQ4ZjE1NGJmMjEtNTdkNi00YzM1LTg4

MzYtNDZlYzNhZDM2NTdkZmQzY2RiNzMtMTU1OC00ZjBmLWFmZGQtM2YyZWU3MDYzZjQwNzVhY

WQ0

ODctZWEyZS00YzM2LWEyZmUtOGVkMmNjMGNmZGM3MTg5YWUyZGQtMzg4MS00M2NkLWI2NDEtZ

jNh

Zjk2OGIxZDU2YzkzODZlYTUtNjljNC00ZmIyLTkzYTQtMzVlOTFlNjdlY2E0MDcwNWExMWItM

mE4

Yy00OTljLTg2NmItMjE3ZTZmMmIyY2NiNzI2MjAwZWQtMjI0NC00YzJhLWJiMGItOTczMDJkM

zIx

NGFkY2QyZmVhODItNmUyOS00MWNjLWFlODItNzdmNzRhYmQ4NGU5ZGQ3ZjY3NzktZjgwYi00M

zlk

LTlmNDgtOTNlNTIwYzIzZDcy

# Java 8 Interview Questions

Dear readers, these **Java 8 Interview Questions** have been designed specially to get you acquainted with the nature of questions you may encounter during your interview for the subject of **Java 8 Language**. As per my experience good interviewers hardly plan to ask any particular question during your interview, normally questions start with some basic concept of the subject and later they continue based on further discussion and what you answer −

What are the new features introduced in JAVA 8?

There are dozens of features added to Java 8, the most significant ones are mentioned below −

* **Lambda expression** − Adds functional processing capability to Java.
* **Method references** − Referencing functions by their names instead of invoking them directly. Using functions as parameter.
* **Default method** − Interface to have default method implementation.
* **New tools** − New compiler tools and utilities are added like 'jdeps' to figure out dependencies.
* **Stream API** − New stream API to facilitate pipeline processing.
* **Date Time API** − Improved date time API.
* **Optional** − Emphasis on best practices to handle null values properly.
* **Nashorn, JavaScript Engine** − A Java-based engine to execute JavaScript code.

Along with these new featuers, lots of feature enhancements are done under-the-hood, at both compiler and JVM level.

How will you sort a list of string using Java 8 lambda expression?

Following code sorts a list of string using Java 8 lambda expression:

//sort using java 8

private void sortUsingJava8(List<String> names){

Collections.sort(names, (s1, s2) -> s1.compareTo(s2));

}

What are the characteristics of a Java 8 lambda expression?

A lambda expression is characterized by the following syntax -

parameter −> expression body

Following are the important characteristics of a lambda expression −

* **Optional type declaration** − No need to declare the type of a parameter. The compiler can inference the same from the value of the parameter.
* **Optional parenthesis around parameter** − No need to declare a single parameter in parenthesis. For multiple parameters, parentheses are required.
* **Optional curly braces** − No need to use curly braces in expression body if the body contains a single statement.
* **Optional return keyword** − The compiler automatically returns the value if the body has a single expression to return the value. Curly braces are required to indicate that expression returns a value.

Why lambda expression is to be used?

Lambda expressions are used primarily to define inline implementation of a functional interface, i.e., an interface with a single method only. In the above example, we've used various types of lambda expressions to define the operation method of MathOperation interface. Then we have defined the implementation of sayMessage of GreetingService.

Lambda expression eliminates the need of anonymous class and gives a very simple yet powerful functional programming capability to Java.

What kind of variable you can access in an lambda expression??

Using lambda expression, you can refer to final variable or effectively final variable whichisassignedonlyoncewhichisassignedonlyonce. Lambda expression throws a compilation error, if a variable is assigned a value the second time.

What are method references?

Method references help to point to methods by their names. A method reference is described using :: doublecolondoublecolon symbol. A method reference can be used to point the following types of methods −

* Static methods
* Instance methods
* Constructors using new operator TreeSet::newTreeSet::new

Explain the System.out::println expression.

System.out::println method is a static method reference to println method of out object of System class.

What are functional interfaces?

Functional interfaces have a single functionality to exhibit. For example, a Comparable interface with a single method 'compareTo' is used for comparison purpose. Java 8 has defined a lot of functional interfaces to be used extensively in lambda expressions.

What is the purpose of BiConsumer<T,U> functional interface?

It represents an operation that accepts two input arguments, and returns no result.

What is the purpose of BiFunction<T,U,R> functional interface?

It represents a function that accepts two arguments and produces a result.

What is the purpose of BinaryOperator<T> functional interface?

It represents an operation upon two operands of the same type, producing a result of the same type as the operands.

What is the purpose of BiPredicate<T,U> functional interface?

It represents a predicate Boolean−valuedfunctionBoolean−valuedfunction of two arguments.

What is the purpose of BooleanSupplier functional interface?

It represents a supplier of Boolean-valued results.

What is the purpose of Consumer<T> functional interface?

It represents an operation that accepts a single input argument and returns no result.

What is the purpose of DoubleBinaryOperator functional interface?

It represents an operation upon two double-valued operands and producing a double-valued result.

What is the purpose of DoubleConsumer functional interface?

It represents an operation that accepts a single double-valued argument and returns no result.

What is the purpose of DoubleFunction<R> functional interface?

It represents a function that accepts a double-valued argument and produces a result.

What is the purpose of DoublePredicate functional interface?

It represents a predicate Boolean−valuedfunctionBoolean−valuedfunction of one double-valued argument.

What is the purpose of DoubleSupplier functional interface?

It represents a supplier of double-valued results.

What is the purpose of DoubleToIntFunction functional interface?

It represents a function that accepts a double-valued argument and produces an int-valued result.

What is the purpose of DoubleToLongFunction functional interface?

It represents a function that accepts a double-valued argument and produces a long-valued result.

What is the purpose of DoubleUnaryOperator functional interface?

It represents an operation on a single double-valued operand that produces a double-valued result.

What is the purpose of Function<T,R> functional interface?

It represents a function that accepts one argument and produces a result.

What is the purpose of IntBinaryOperator functional interface?

It represents an operation upon two int-valued operands and produces an int-valued result.

What is the purpose of IntConsumer functional interface?

It represents an operation that accepts a single int-valued argument and returns no result.

What is the purpose of IntFunction<R> functional interface?

It represents a function that accepts an int-valued argument and produces a result.

What is the purpose of IntPredicate functional interface?

It represents a predicate Boolean−valuedfunctionBoolean−valuedfunction of one int-valued argument.

What is the purpose of IntSupplier functional interface?

It represents a supplier of int-valued results.

What is the purpose of IntToDoubleFunction functional interface?

It represents a function that accepts an int-valued argument and produces a double-valued result.

What is the purpose of IntToLongFunction functional interface?

It represents a function that accepts an int-valued argument and produces a long-valued result.

What is the purpose of IntUnaryOperator functional interface?

It represents an operation on a single int-valued operand that produces an int-valued result.

What is the purpose of LongBinaryOperator functional interface?

It represents an operation upon two long-valued operands and produces a long-valued result.

What is the purpose of LongConsumer functional interface?

It represents an operation that accepts a single long-valued argument and returns no result.

What is the purpose of LongFunction<R> functional interface?

It represents a function that accepts a long-valued argument and produces a result.

What is the purpose of LongPredicate functional interface?

It represents a predicate Boolean−valuedfunctionBoolean−valuedfunction of one long-valued argument.

What is the purpose of LongSupplier functional interface?

It represents a supplier of long-valued results.

What is the purpose of LongToDoubleFunction functional interface?

It represents a function that accepts a long-valued argument and produces a double-valued result.

What is the purpose of LongToIntFunction functional interface?

It represents a function that accepts a long-valued argument and produces an int-valued result.

What is the purpose of LongUnaryOperator functional interface?

It represents an operation on a single long-valued operand that produces a long-valued result.

What is the purpose of ObjDoubleConsumer<T> functional interface?

It represents an operation that accepts an object-valued and a double-valued argument, and returns no result.

What is the purpose of ObjIntConsumer<T> functional interface?

It represents an operation that accepts an object-valued and an int-valued argument, and returns no result.

What is the purpose of ObjLongConsumer<T> functional interface?

It represents an operation that accepts an object-valued and a long-valued argument, and returns no result.

What is the purpose of Predicate<T> functional interface?

It represents a predicate Boolean−valuedfunctionBoolean−valuedfunction of one argument.

What is the purpose of Supplier<T> functional interface?

It represents a supplier of results.

What is the purpose of ToDoubleBiFunction<T,U> functional interface?

It represents a function that accepts two arguments and produces a double-valued result.

What is the purpose of ToDoubleFunction<T> functional interface?

It represents a function that produces a double-valued result.

What is the purpose of ToIntBiFunction<T,U> functional interface?

It represents a function that accepts two arguments and produces an int-valued result.

What is the purpose of ToIntFunction<T> functional interface?

It represents a function that produces an int-valued result.

What is the purpose of ToLongBiFunction<T,U> functional interface?

It represents a function that accepts two arguments and produces a long-valued result.

What is the purpose of ToLongFunction<T> functional interface?

It represents a function that produces a long-valued result.

What is the purpose of UnaryOperator<T> functional interface?

It represents an operation on a single operand that produces a result of the same type as its operand.

What are default methods?

With java 8, an interface can have default implementation of a function in interfaces.

What are static default methods?

An interface can also have static helper methods from Java 8 onwards.

public interface vehicle {

default void print(){

System.out.println("I am a vehicle!");

}

static void blowHorn(){

System.out.println("Blowing horn!!!");

}

}

How will you call a default method of an interface in a class?

Using super keyword along with interface name.

interface Vehicle {

default void print(){

System.out.println("I am a vehicle!");

}

}

class Car implements Vehicle {

public void print(){

Vehicle.super.print();

}

}

How will you call a static method of an interface in a class?

Using name of the interface.

interface Vehicle {

static void blowHorn(){

System.out.println("Blowing horn!!!");

}

}

class Car implements Vehicle {

public void print(){

Vehicle.blowHorn();

}

}

What is streams in Java 8?

Stream represents a sequence of objects from a source, which supports aggregate operations.

What is stream pipelining in Java 8?

Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.

What is the difference between Collections and Stream in Java8 ?

Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

What is the purpose of forEach method of stream in java 8?

Stream has provided a new method 'forEach' to iterate each element of the stream.

How will you print 10 random numbers using forEach of java 8?

The following code segment shows how to print 10 random numbers using forEach.

Random random = new Random();

random.ints().limit(10).forEach(System.out::println);

What is the purpose of map method of stream in java 8?

The 'map' method is used to map each element to its corresponding result.

How will you print unique squares of numbers in java 8?

The following code segment prints unique squares of numbers using map.

List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

//get list of unique squares

List<Integer> squaresList = numbers.stream().map( i -> i\*i).distinct().collect(Collectors.toList());

What is the purpose of filter method of stream in java 8?

The 'filter' method is used to eliminate elements based on a criteria.

How will you print count of empty strings in java 8?

The following code segment prints a count of empty strings using filter.

List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string

int count = strings.stream().filter(string −> string.isEmpty()).count();

What is the purpose of limit method of stream in java 8?

The 'limit' method is used to reduce the size of the stream.

How will you print 10 random numbers in java 8?

The following code segment shows how to print 10 random numbers.

Random random = new Random();

random.ints().limit(10).forEach(System.out::println);

What is the purpose of sorted method of stream in java 8?

The 'sorted' method is used to sort the stream.

How will you print 10 random numbers in a sorted order in java 8?

The following code segment shows how to print 10 random numbers in a sorted order.

Random random = new Random();

random.ints().limit(10).sorted().forEach(System.out::println);

What is Parallel Processing in Java 8?

parallelStream is the alternative of stream for parallel processing. Take a look at the following code segment that prints a count of empty strings using parallelStream.

List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string

int count = strings.parallelStream().filter(string −> string.isEmpty()).count();

//It is very easy to switch between sequential and parallel streams.

What are collectors in Java 8?

Collectors are used to combine the result of processing on the elements of a stream. Collectors can be used to return a list or a string.

List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());

System.out.println("Filtered List: " + filtered);

String mergedString = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.joining(", "));

System.out.println("Merged String: " + mergedString);

What are Statistics collectors in Java 8?

With Java 8, statistics collectors are introduced to calculate all statistics when stream processing is being done.

How will you get the highest number present in a list using Java 8?

Following code will print the highest number present in a list.

List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

IntSummaryStatistics stats = integers.stream().mapToInt((x) −> x).summaryStatistics();

System.out.println("Highest number in List : " + stats.getMax());

How will you get the lowest number present in a list using Java 8?

Following code will print the highest number present in a list.

List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

IntSummaryStatistics stats = integers.stream().mapToInt((x) −> x).summaryStatistics();

System.out.println("Lowest number in List : " + stats.getMin());

How will you get the sum of all numbers present in a list using Java 8?

Following code will print the sum of all numbers present in a list.

List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

IntSummaryStatistics stats = integers.stream().mapToInt((x) −> x).summaryStatistics();

System.out.println("Sum of all numbers : " + stats.getSum());

How will you get the average of all numbers present in a list using Java 8?

Following code will print the average of all numbers present in a list.

List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

IntSummaryStatistics stats = integers.stream().mapToInt((x) −> x).summaryStatistics();

System.out.println("Average of all numbers : " + stats.getAverage());

What is Optional in Java8?

Optional is a container object which is used to contain not-null objects. Optional object is used to represent null with absent value. This class has various utility methods to facilitate code to handle values as 'available' or 'not available' instead of checking null values. It is introduced in Java 8 and is similar to what Optional is in Guava.

What is Nashorn in Java8?

With Java 8, Nashorn, a much improved javascript engine is introduced, to replace the existing Rhino. Nashorn provides 2 to 10 times better performance, as it directly compiles the code in memory and passes the bytecode to JVM. Nashorn uses invokedynamics feature, introduced in Java 7 to improve performance.

What is jjs in JAVA8?

For Nashorn engine, JAVA 8 introduces a new command line tool, jjs, to execute javascript codes at console.

Can you execute javascript code from java 8 code base?

Yes! Using ScriptEngineManager, JavaScript code can be called and interpreted in Java.

What is local datetime API in JAVA8?

Local − Simplified date-time API with no complexity of timezone handling.

What is zoned datetime API in JAVA8?

Zoned − Specialized date-time API to deal with various timezones.

What is chromounits in java8?

java.time.temporal.ChronoUnit enum is added in Java 8 to replace the integer values used in old API to represent day, month, etc.

How will you get the current date using local datetime api of java8?

Following code gets the current date using local datetime api −

//Get the current date

LocalDate today = LocalDate.now();

System.out.println("Current date: " + today);

How will you add 1 week to current date using local datetime api of java8?

Following code adds 1 week to current date using local datetime api −

//add 1 week to the current date

LocalDate today = LocalDate.now();

LocalDate nextWeek = today.plus(1, ChronoUnit.WEEKS);

System.out.println("Next week: " + nextWeek);

How will you add 1 month to current date using local datetime api of java8?

Following code adds 1 month to current date using local datetime api:

//add 1 month to the current date

LocalDate today = LocalDate.now();

LocalDate nextMonth = today.plus(1, ChronoUnit.MONTHS);

System.out.println("Next month: " + nextMonth);

How will you add 1 year to current date using local datetime api of java8?

Following code adds 1 year to current date using local datetime api −

//add 1 year to the current date

LocalDate today = LocalDate.now();

LocalDate nextYear = today.plus(1, ChronoUnit.YEARS);

System.out.println("Next year: " + nextYear);

How will you add 10 years to current date using local datetime api of java8?

Following code adds 10 years to current date using local datetime api −

//add 10 years to the current date

LocalDate today = LocalDate.now();

LocalDate nextDecade = today.plus(1, ChronoUnit.DECADES);

System.out.println("Date after ten year: " + nextDecade);

How will you get next tuesday using java8?

Following code gets next tuesday using java8 −

//get the next tuesday

LocalDate today = LocalDate.now();

LocalDate nextTuesday = today.with(TemporalAdjusters.next(DayOfWeek.TUESDAY));

System.out.println("Next Tuesday on : " + nextTuesday);

How will you get second saturday of next month using java8?

Following code gets second saturday of next month using java8 −

//get the second saturday of next month

LocalDate firstInYear = LocalDate.of(date1.getYear(),date1.getMonth(), 1);

LocalDate secondSaturday = firstInYear.with(TemporalAdjusters.nextOrSame(DayOfWeek.SATURDAY)).with(TemporalAdjusters.next(DayOfWeek.SATURDAY));

System.out.println("Second Saturday on : " + secondSaturday);

How will you get the instant of current date in terms of milliseconds using java8?

Following code gets the instant of current date in terms of milliseconds −

//Get the instant of current date in terms of milliseconds

Instant now = currentDate.toInstant();

How will you get the instant of local date time using time in of milliseconds using java8?

Following code gets the instant of local date time using time in of milliseconds −

Instant now = currentDate.toInstant();

ZoneId currentZone = ZoneId.systemDefault();

LocalDateTime localDateTime = LocalDateTime.ofInstant(now, currentZone);

System.out.println("Local date: " + localDateTime);

How will you get the instant of zoned date time using time in of milliseconds using java8?

Following code gets the instant of zoned date time using time in of milliseconds −

Instant now = currentDate.toInstant();

ZoneId currentZone = ZoneId.systemDefault();

ZonedDateTime zonedDateTime = ZonedDateTime.ofInstant(now, currentZone);

System.out.println("Zoned date: " + zonedDateTime);

Which class implements a decoder for decoding byte data using the Base64 encoding scheme in Java8?

static class Base64.Decoder − This class implements a decoder for decoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045.

Which class implements an encoder for encoding byte data using the Base64 encoding scheme in Java8?

static class Base64.Encoder − This class implements an encoder for encoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045.

How will you create a Base64 decoder?

getDecoder method of Base64 class returns a Base64.Decoder that decodes using the Basic type base64 encoding scheme.

How will you create a Base64 encoder?

getEncoder method of Base64 class returns a Base64.Encoder that encodes using the Basic type base64 encoding scheme.

How will you create a Base64 decoder that decodes using the MIME type base64 encoding scheme?

getMimeDecoder method of Base64 class returns a Base64.Decoder that decodes using the MIME type base64 decoding scheme.

How will you create a Base64 encoder that encodes using the MIME type base64 encoding scheme?

getMimeEncoder method of Base64 class returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme.

How will you create a Base64 decoder that decodes using the URL and Filename safe type base64 encoding scheme?

getUrlDecoder method of Base64 class returns a Base64.Decoder that decodes using the URL and Filename safe type base64 encoding scheme.

How will you create a Base64 encoder that encodes using the URL and Filename safe type base64 encoding scheme?

getUrlEncoder method of Base64 class returns a Base64.Encoder that encodes using the URL and Filename safe type base64 encoding scheme.

## What is Next?

Further you can go through your past assignments you have done with the subject and make sure you are able to speak confidently on them. If you are fresher then interviewer does not expect you will answer very complex questions, rather you have to make your basics concepts very strong.

Second it really doesn't matter much if you could not answer few questions but it matters that whatever you answered, you must have answered with confidence. So just feel confident during your interview. We at tutorialspoint wish you best luck to have a good interviewer and all the very best for your future endeavor. Cheers :-)

### Java 8 Interview Questions

1. Why do we need change to Java again?
2. Java SE 8 New Features?
3. Advantages of Java SE 8 New Features?
4. What is Lambda Expression?
5. What are the three parts of a Lambda Expression? What is the type of Lambda Expression?
6. What is a Functional Interface? What is SAM Interface?
7. Is is possible to define our own Functional Interface? What is @FunctionalInterface? What are the rules to define a Functional Interface?
8. Is @FunctionalInterface annotation mandatory to define a Functional Interface? What is the use of @FunctionalInterface annotation? Why do we need Functional Interfaces in Java?
9. When do we go for Java 8 Stream API? Why do we need to use Java 8 Stream API in our projects?
10. Explain Differences between Collection API and Stream API?
11. What is Spliterator in Java SE 8?Differences between Iterator and Spliterator in Java SE 8?
12. What is Optional in Java 8? What is the use of Optional?Advantages of Java 8 Optional?
13. What is Type Inference? Is Type Inference available in older versions like Java 7 and Before 7 or it is available only in Java SE 8?

### Java 8 Interview Questions and Answers

In this section, we will pickup each question from previous section and answer it with in-detailed description. If you need any more information and examples, please go through previous Java SE 8 posts available in JournalDEV.

### Why do we need change to Java again?

Oracle Corporation has introduced a lot of new concepts in Java SE 8 to introduce the following benefits:

* **To Utilize Current Multi-Core CPUs Efficiently**

Recently, we can observe drastic changes in Hardware. Now-a-days, all systems are using Multi-Core CPUs(2,4,8,16-Core etc.) to deploy and run their Applications. We need new Programming Constructs in Java to utilize these Multi-Core Processors efficiently to develop Highly Concurrently and Highly Scalable applications.

* **To Utilize FP Features**

Oracle Corporation has introduced a lot of FP(Functional Programming) concepts as part of Java SE 8 to utilize the advantages of FP.

### Java SE 8 New Features?

* Lambda Expressions
* Functional Interfaces
* Stream API
* Date and Time API
* Interface Default Methods and Static Methods
* Spliterator
* Method and Constructor References
* Collections API Enhancements
* Concurrency Utils Enhancements
* Fork/Join Framework Enhancements
* Internal Iteration
* Parallel Array and Parallel Collection Operations
* Optional
* Type Annotations and Repeatable Annotations
* Method Parameter Reflection
* Base64 Encoding and Decoding
* IO and NIO2 Enhancements
* Nashorn JavaScript Engine
* javac Enhancements
* JVM Changes
* Java 8 Compact Profiles: compact1,compact2,compact3
* JDBC 4.2
* JAXP 1.6
* Java DB 10.10
* Networking
* Security Changes

### Advantages of Java SE 8 New Features?

We can get the following benefits from Java SE 8 New Features:

* More Concise and Readable code
* More Reusable code
* More Testable and Maintainable Code
* Highly Concurrent and Highly Scalable Code
* Write Parallel Code
* Write Database Like Operations
* Better Performance Applications
* More Productive code

### What is Lambda Expression?

Lambda Expression is an anonymous function which accepts a set of input parameters and returns results.

Lambda Expression is a block of code without any name, with or without parameters and with or without results. This block of code is executed on demand.

### What are the three parts of a Lambda Expression? What is the type of Lambda Expression?

A Lambda Expression contains 3 parts:

* Parameter List

A Lambda Expression can contain zero or one or more parameters. It is optional.

* Lambda Arrow Operator

“->” is known as Lambda Arrow operator. It separates parameters list and body.

* Lambda Expression Body

The type of “Journal Dev” is java.lang.String. The type of “true” is Boolean. In the same way, what is the type of a Lambda Expression?  
The Type of a Lambda Expression is a Functional Interface.

Example:- What is the type of the following Lambda Expression?

|  |  |
| --- | --- |
| 1 | () -> System.out.println("Hello World"); |

This Lambda Expression does not have parameters and does return any results. So it’s type is “java.lang.Runnable” Functional Interface.

### What is a Functional Interface? What is SAM Interface?

A Functional Interface is an interface, which contains one and only one abstract method. Functional Interface is also know as SAM Interface because it contains only one abstract method.

SAM Interface stands for Single Abstract Method Interface. Java SE 8 API has defined many Functional Interfaces.

### Is is possible to define our own Functional Interface? What is @FunctionalInterface? What are the rules to define a Functional Interface?

Yes, it is possible to define our own Functional Interfaces. We use Java SE 8’s @FunctionalInterface annotation to mark an interface as Functional Interface.

We need to follow these rules to define a Functional Interface:

* Define an interface with one and only one abstract method.
* We cannot define more than one abstract method.
* Use @FunctionalInterface annotation in interface definition.
* We can define any number of other methods like Default methods, Static methods.
* If we override java.lang.Object class’s method as an abstract method, which does not count as an abstract method.

### Is @FunctionalInterface annotation mandatory to define a Functional Interface? What is the use of @FunctionalInterface annotation? Why do we need Functional Interfaces in Java?

It is not mandatory to define a Functional Interface with @FunctionalInterface annotation. If we don’t want, We can omit this annotation. However, if we use it in Functional Interface definition, Java Compiler forces to use one and only one abstract method inside that interface.

Why do we need Functional Interfaces? The type of a Java SE 8’s Lambda Expression is a Functional Interface. Whereever we use Lambda Expressions that means we are using Functional Interfaces.

### When do we go for Java 8 Stream API? Why do we need to use Java 8 Stream API in our projects?

When our Java project wants to perform the following operations, it’s better to use Java 8 Stream API to get lot of benefits:

* When we want perform Database like Operations. For instance, we want perform groupby operation, orderby operation etc.
* When want to Perform operations Lazily.
* When we want to write Functional Style programming.
* When we want to perform Parallel Operations.
* When want to use Internal Iteration
* When we want to perform Pipelining operations.
* When we want to achieve better performance.

### Explain Differences between Collection API and Stream API?

|  |  |  |
| --- | --- | --- |
| S.NO. | COLLECTION API | STREAM API |
| 1. | It’s available since Java 1.2 | It is introduced in Java SE8 |
| 2. | It is used to store Data(A set of Objects). | It is used to compute data(Computation on a set of Objects). |
| 3. | We can use both Spliterator and Iterator to iterate elements. | We can use both Spliterator and Iterator to iterate elements. |
| 4. | It is used to store limited number of Elements. | It is used to store either Limited or Infinite Number of Elements. |
| 5. | Typically, it uses Internal Iteration concept to iterate Elements. | It uses External Iteration to iterate Elements. |
| 6. | Collection Object is constructed Eagerly. | Stream Object is constructed Lazily. |
| 7. | We add elements to Collection object only after it is computed completely. | We can add elements to Stream Object without any prior computation. That means Stream objects are computed on-demand. |
| 8. | We can iterate and consume elements from a Collection Object at any number of times. | We can iterate and consume elements from a Stream Object only once. |

### What is Spliterator in Java SE 8?Differences between Iterator and Spliterator in Java SE 8?

Spliterator stands for Splitable Iterator. It is newly introduced by Oracle Corporation as part Java SE 8.  
Like Iterator and ListIterator, It is also one of the Iterator interface.

|  |  |  |
| --- | --- | --- |
| S.NO. | SPLITERATOR | ITERATOR |
| 1. | It is introduced in Java SE 8. | It is available since Java 1.2. |
| 2. | Splitable Iterator | Non-Splitable Iterator |
| 3. | It is used in Stream API. | It is used for Collection API. |
| 4. | It uses Internal Iteration concept to iterate Streams. | It uses External Iteration concept to iterate Collections. |
| 5. | We can use Spliterator to iterate Streams in Parallel and Sequential order. | We can use Spliterator to iterate Collections only in Sequential order. |
| 6. | We can get Spliterator by calling spliterator() method on Stream Object. | We can get Iterator by calling iterator() method on Collection Object. |
| 7. | Important Method: tryAdvance() | Important Methods: next(), hasNext() |

### What is Optional in Java 8? What is the use of Optional?Advantages of Java 8 Optional?

**Optional:**  
Optional is a final Class introduced as part of Java SE 8. It is defined in java.util package.

It is used to represent optional values that is either exist or not exist. It can contain either one value or zero value. If it contains a value, we can get it. Otherwise, we get nothing.

It is a bounded collection that is it contains at most one element only. It is an alternative to “null” value.

**Main Advantage of Optional is:**

* It is used to avoid null checks.
* It is used to avoid “NullPointerException”.

### What is Type Inference? Is Type Inference available in older versions like Java 7 and Before 7 or it is available only in Java SE 8?

Type Inference means determining the Type by compiler at compile-time.

It is not new feature in Java SE 8. It is available in Java 7 and before Java 7 too.

**Before Java 7:-**  
Let us explore Java arrays. Define a String of Array with values as shown below:

|  |  |
| --- | --- |
| 1 | String str[] = { "Java 7", "Java 8", "Java 9" }; |

Here we have assigned some String values at right side, but not defined it’s type. Java Compiler automatically infers it’s type and creates a String of Array.

**Java 7:-**  
Oracle Corporation has introduced “Diamond Operator” new feature in Java SE 7 to avoid unnecessary Type definition in Generics.

|  |  |
| --- | --- |
| 1 | Map<String,List<Customer>> customerInfoByCity = new HashMap<>(); |

Here we have not defined Type information at right side, simply defined Java SE 7’s Diamond Operator “”.

**Java SE 8:-**  
Oracle Corporation has enhanced this Type Inference concept a lot in Java SE 8. We use this concept to define Lambda Expressions, Functions, Method References etc.

|  |  |
| --- | --- |
| 1 | Integer add = (a,b) -> a + b; |

Here Java Compiler observes the type definition available at left-side and determines the type of Lambda Expression parameters a and b is Integer.

That’s it about Java 8 Interview Questions.

I have discussed some Java SE 8 Interview Questions in this post and will discuss some more Java SE 8 Interview Questions in my coming posts.

Please drop me a comment if you like my post or have any issues/suggestions

### Java 8 - Interview Questions and Answers on Java 8

**Q1.  What are new features introduced with Java 8 ?**  
Ans. Lambda Expressions , Interface Default and Static Methods , Method Reference , Parameters Name , Optional , Streams, Concurrency.

**Q2.  What is a Lambda Expression ? What's its use ?**  
  
Ans. Its an anonymous method without any declaration. Lambda Expression are useful to write shorthand Code and hence saves the effort of writing lengthy Code.    It promotes Developer productivity, Better Readable and Reliable code.  
  
**Q3.  Difference between Predicate, Supplier and Consumer ?**  
  
Ans. Predicate represents an anonymous function that accepts one argument and produces a result.  
  
Supplier represents an anonymous function that accepts no argument and produces a result.  
  
Consumer represents an anonymous function that accepts an argument and produces no result.  
  
**Q4.  What does the following lambda expression means ?**  
  
helloJava8 ( x-> x%2 )  
  
  
Ans. helloJava8 receives an Integer as argument and then returns the modulus of that Integer.  
  
**Q5.  What are Default Methods ?**  
  
Ans. With Java 8, We can provide method definitions in the Interfaces that gets carried down the classes implementing that interface in case they are not overridden by the Class. Keyword "default" is used to mark the default method.  
  
**Q6.  Can we have a default method definition in the interface without specifying the keyword "default" ?**  
  
Ans. No. Compiler complains that its an abstract method and hence shouldn't have the body.  
  
**Q7.  Can a class implement two Interfaces having default method with same name and signature ?**  
  
public interface DefaultMethodInterface {  
    default public void defaultMethod(){  
       System.out.println("DefaultMethodInterface");          
    }  
}  
  
public interface DefaultMethodInterface2 {            
        default public void defaultMethod(){  
               System.out.println("DefaultMethodInterface2");          
        }  
}  
  
public class HelloJava8 implements DefaultMethodInterface,DefaultMethodInterface2 {  
   public static void main(String[] args){     
           DefaultMethodInterface defMethIn = new HelloJava8();  
           defMethIn.defaultMethod();  
    }  
}  
  
Ans. No. Compiler gives error saying "Duplicate Default Methods"  
  
**Q8.  What If we make the method as abstract in another Interface ?**  
  
public interface DefaultMethodInterface {  
    default public void defaultMethod(){  
       System.out.println("DefaultMethodInterface");          
    }  
}  
  
public interface DefaultMethodInterface2 {            
        public void defaultMethod(){  
               System.out.println("DefaultMethodInterface2");          
        }  
}  
  
public class HelloJava8 implements DefaultMethodInterface,DefaultMethodInterface2 {  
   public static void main(String[] args){     
           DefaultMethodInterface defMethIn = new HelloJava8();  
           defMethIn.defaultMethod();  
    }  
}  
  
Ans. Even then the Compiler will give error saying that there is a conflict.  
  
**Q9.  What if we override the conflicting method in the Class ?**  
  
public interface DefaultMethodInterface {  
    default public void defaultMethod(){  
       System.out.println("DefaultMethodInterface");          
    }  
}  
  
public interface DefaultMethodInterface2 {            
        default public void defaultMethod(){  
               System.out.println("DefaultMethodInterface2");          
        }  
}  
  
public class HelloJava8 implements DefaultMethodInterface,DefaultMethodInterface2 {  
   public static void main(String[] args){     
           DefaultMethodInterface defMethIn = new HelloJava8();  
           defMethIn.defaultMethod();  
    }  
  
   public void defaultMethod(){  
       System.out.println("HelloJava8");   
   }  
}  
  
Ans. There won't be any error and upon execution the overriding class method will be executed.   
  
**Q10.  What will happen if there is a default method conflict as mentioned above and we have specified the same signature method in the base class instead of overriding in the existing class ?**  
  
Ans. There won't be any problem as the Base class method will have precedence over the Interface Default methods.  
  
**Q11.  If there is a conflict between Base Class Method definition and Interface Default method definition, Which definition is Picked ?**  
  
Ans. Base Class Definition.  
  
**Q12.  If a method definition has been specified in Class , its Base Class , and the interface which the class is implementing, Which definition will be picked if we try to access it using Interface Reference  and Class object ?**  
  
Ans. Class method definition is overriding both the definitions and hence will be picked.  
  
**Q13.  If a method definition has been specified in the Base Class and the interface which the class is implementing, Which definition will be picked if we try to access it using Interface Reference and Class object ?**  
  
Ans. Base Class Definition will have precedence over the Interface Default method definition.  
  
**Q14.  Can we use static method definitions in Interfaces ?**  
  
Ans. Yes, Effective Java 8.  
  
**Q15.  Can we access Interface static method using Interface references ?**  
  
Ans. No, only using Interface Name.  
  
**Q16.  Can we have default method with same name and signature in the derived Interface as the static method in base Interface and vice versa ?**  
  
Ans. Yes , we can do that as static methods are not accessible using references and hence cannot lead to conflict. We cannot do inverse as Default methods cannot be overridden with the static methods in derived interface.  
  
**Q17.  Name few Java 8 annotations ?**  
  
Ans.   
  
@FunctionalInterface annotation, introduced in Java SE 8, indicates that the type declaration is intended to be a functional interface, as defined by the Java Language Specification.  
  
@Repeatable annotation, introduced in Java SE 8, indicates that the marked annotation can be applied more than once to the same declaration or type use. For more information, see Repeating Annotations.

**Q18.  What is the @FunctionalInterface annotation ?**  
  
Ans. This is an informative annotation that specify that the interface is a functional interface. A Function Interface has only one abstract method and many default methods. Compiler generates an error if the interface specified with the annotation doesn't abide by the specifications for functional interface.  
  
**Q19.  Difference between final and effectively final ? Why is effectively final even required ?**  
  
Ans. Final variable means a variable that has been declared final and hence cannot be changed after initialization.  
  
Effective final means a variable that has not been declared final but haven't been reassigned the value after initialization.  
  
First is the regulation that restricts the reassignment and will raise a compilation error if we try to do so. Second is the outcome without the restriction.  
  
Effective Final is the eventual treatment of the variable that is required for many features. For eq - Java 8 requires that  
  
local variables referenced from a lambda expression must be final or effectively final  
  
It means all local referenced from lambda expressions must be such that their value shouldn't be changed after initialization whether declared final or not.  
  
**Q20.  Difference between DoubleSummaryStatistics , IntSummaryStatistics and LongSummaryStatistics ?**  
  
Ans. They all does the same task i.e to compute statistical information on the stream of data. They differ by the way they store the statistical information as they expect a different data type of the values being used.   
  
IntSummaryStatistics and LongSummaryStatistics expect non floating point values and hence stores the statistical information like min,max and sum as non floating values ( int or long ) whereas DoubleSummaryStatistics stores these information as floating value.  
  
**Q21.  What is StringJoiner ?**  
  
Ans. StringJoiner is a util method to construct a string with desired delimiter. This has been introduced with wef from Java 8.  
  
Sample Code  
  
StringJoiner strJoiner = new StringJoiner(".");  
strJoiner.add("Buggy").add("Bread");  
System.out.println(strJoiner); // prints Buggy.Bread  
  
**Q22.  What is the use of Optional ?**  
  
Ans. Optional is a good way to protect application from runtime nullPointerException in case the the absent value has been represented as null. So basically Optional class provides the type checking during compile time and hence will never result in NPE.  
  
For ex -  
  
List<Optional<Employee>> intList = new ArrayList<Optional<Employee>>();  
intList.add(Optional.empty());  
intList.add(Optional.of(new Employee("abc")));  
intList.add(Optional.of(new Employee("xyz")));  
intList.add(Optional.of(new Employee("123")));  
System.out.println(intList.get(0).getName());  
  
So Now , even when the first list element is empty, this code will never throw an NullPointerException.  
  
**Q23.  Name few "Optional" classes introduced with Java 8 ?**  
  
Ans.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Type** | **Parent** | **Since** | **Detail** |
|  |  |  |  |  |
| Optional | Class | java.lang.Object | 1.8 | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Optional.html) |
| OptionalDataException | Class | java.io.ObjectStreamException | 1.1 | [Link](http://docs.oracle.com/javase/8/docs/api/java/io/OptionalDataException.html) |
| OptionalDouble | Class | java.lang.Object | 1.8 | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/OptionalDouble.html) |
| OptionalInt | Class | java.lang.Object | 1.8 | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/OptionalInt.html) |
| OptionalLong | Class | java.lang.Object | 1.8 | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/OptionalLong.html) |

**Q24.  What are the changes in Java.io in Java 8 ?**  
  
Ans

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Parent** | **JavaDoc** |
|  |  |  |  |
| UncheckedIOException | Class | java.lang.RuntimeException | [Link](http://docs.oracle.com/javase/8/docs/api/java/io/UncheckedIOException.html) |

### Q25.  Name few Java Util classes introduced with Java 8 ? Ans. Migrating to Java 8 - New Classes / Interfaces introduced within java.util

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Parent** | **JavaDoc** |
|  |  |  |  |
| PrimitiveIterator | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/PrimitiveIterator.html) |
| PrimitiveIterator.OfDouble | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/PrimitiveIterator.OfDouble.html) |
| PrimitiveIterator.OfInt | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/PrimitiveIterator.OfInt.html) |
| PrimitiveIterator.OfLong | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/PrimitiveIterator.OfLong.html) |
| Spliterator | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html) |
| Spliterator.OfDouble | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.OfDouble.html) |
| Spliterator.OfInt | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.OfInt.html) |
| Spliterator.OfLong | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.OfLong.html) |
| Spliterator.OfPrimitive | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.OfPrimitive.html) |
| Base64 | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) |
| Base64.Decoder | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html) |
| Base64.Encoder | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) |
| Calendar.Builder | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Calendar.Builder.html) |
| DoubleSummaryStatistics | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/DoubleSummaryStatistics.html) |
| IntSummaryStatistics | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/IntSummaryStatistics.html) |
| Locale.LanguageRange | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Locale.LanguageRange.html) |
| LongSummaryStatistics | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/LongSummaryStatistics.html) |
| Optional | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Optional.html) |
| OptionalDouble | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/OptionalDouble.html) |
| OptionalInt | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/OptionalInt.html) |
| OptionalLong | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/OptionalLong.html) |
| Spliterators | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Spliterators.html) |
| Spliterators.AbstractDoubleSpliterator | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Spliterators.AbstractDoubleSpliterator.html) |
| Spliterators.AbstractIntSpliterator | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Spliterators.AbstractIntSpliterator.html) |
| Spliterators.AbstractLongSpliterator | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Spliterators.AbstractLongSpliterator.html) |
| Spliterators.AbstractSpliterator | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Spliterators.AbstractSpliterator.html) |
| SplittableRandom | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/SplittableRandom.html) |
| StringJoiner | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/StringJoiner.html) |
| Locale.FilteringMode | Class | java.lang.Enum | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Locale.FilteringMode.html) |
| CompletableFuture.AsynchronousCompletionTask | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.AsynchronousCompletionTask.html) |
| CompletionStage | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html) |
| CompletableFuture | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html) |
| ConcurrentHashMap.KeySetView | Class | ConcurrentHashMap.KeySetView | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.KeySetView.html) |
| CountedCompleter | Class | concurrent.ForkJoinTask | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CountedCompleter.html) |
| CompletionException | Class | java.lang.RuntimeException | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionException.html) |
| Spliterator | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html) |
| DoubleAccumulator | Class | java.lang.Number | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/DoubleAccumulator.html) |
| DoubleAdder | Class | java.lang.Number | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/DoubleAdder.html) |
| LongAccumulator | Class | java.lang.Number | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAccumulator.html) |
| LongAdder | Class | java.lang.Number | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html) |
| StampedLock | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html) |
| BiConsumer | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html) |
| BiFunction | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html) |
| BinaryOperator | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/BinaryOperator.html) |
| BiPredicate | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html) |
| BooleanSupplier | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/BooleanSupplier.html) |
| Consumer | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) |
| DoubleBinaryOperator | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleBinaryOperator.html) |
| DoubleConsumer | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleConsumer.html) |
| DoubleFunction | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleFunction.html) |
| DoublePredicate | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/DoublePredicate.html) |
| DoubleSupplier | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleSupplier.html) |
| DoubleToIntFunction | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleToIntFunction.html) |
| DoubleToLongFunction | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleToLongFunction.html) |
| DoubleUnaryOperator | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleUnaryOperator.html) |
| Function | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html) |
| IntBinaryOperator | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/IntBinaryOperator.html) |
| IntConsumer | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/IntConsumer.html) |
| IntFunction | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/IntFunction.html) |
| IntPredicate | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/IntPredicate.html) |
| IntSupplier | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/IntSupplier.html) |
| IntToDoubleFunction | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/IntToDoubleFunction.html) |
| IntToLongFunction | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/IntToLongFunction.html) |
| IntUnaryOperator | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/IntUnaryOperator.html) |
| LongBinaryOperator | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/LongBinaryOperator.html) |
| LongConsumer | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/LongConsumer.html) |
| LongFunction | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/LongFunction.html) |
| LongPredicate | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/LongPredicate.html) |
| LongSupplier | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/LongSupplier.html) |
| LongToDoubleFunction | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/LongToDoubleFunction.html) |
| LongToIntFunction | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/LongToIntFunction.html) |
| LongUnaryOperator | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/LongUnaryOperator.html) |
| ObjDoubleConsumer | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/ObjDoubleConsumer.html) |
| ObjIntConsumer | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/ObjIntConsumer.html) |
| ObjLongConsumer | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/ObjLongConsumer.html) |
| Predicate | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html) |
| Supplier | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html) |
| ToDoubleBiFunction | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/ToDoubleBiFunction.html) |
| ToDoubleFunction | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/ToDoubleFunction.html) |
| ToIntBiFunction | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/ToIntBiFunction.html) |
| ToIntFunction | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/ToIntFunction.html) |
| ToLongBiFunction | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/ToLongBiFunction.html) |
| ToLongFunction | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/ToLongFunction.html) |
| UnaryOperator | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/function/UnaryOperator.html) |
| ResourceBundleControlProvider | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/spi/ResourceBundleControlProvider.html) |
| CalendarDataProvider | Class | java.util.spi.LocaleServiceProvider | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/spi/CalendarDataProvider.html) |
| CalendarNameProvider | Class | java.util.spi.LocaleServiceProvider | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/spi/CalendarNameProvider.html) |
| BaseStream | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html) |
| Collector | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html) |
| DoubleStream | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/stream/DoubleStream.html) |
| DoubleStream.Builder | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/stream/DoubleStream.Builder.html) |
| IntStream | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html) |
| IntStream.Builder | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.Builder.html) |
| LongStream | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/stream/LongStream.html) |
| LongStream.Builder | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/stream/LongStream.Builder.html) |
| Stream | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html) |
| Stream.Builder | Interface |  | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.Builder.html) |
| Collectors | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html) |
| StreamSupport | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/stream/StreamSupport.html) |
| Spliterators | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/Spliterators.html) |
| Adler32 | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/zip/Adler32.html) |
| CRC32 | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/zip/CRC32.html) |
| ZipEntry | Class | java.lang.Object | [Link](http://docs.oracle.com/javase/8/docs/api/java/util/zip/ZipEntry.html) |

**Q26.  Name few java.lang classes introduced with Java 8 ?**  
  
Ans. **Classes**  
  
Native  
Parameter  
MalformedParametersException  
  
**Interfaces**  
  
MethodHandleInfo  
AnnotatedArrayType  
AnnotatedParameterizedType  
AnnotatedType  
AnnotatedTypeVariable  
AnnotatedWildcardType  
  
**Annotation Type**  
  
FunctionalInterface  
Repeatable  
Executable**Q27.  Which keyword specify that a variable is effectively final ?**  
  
 a. final  
 b. No Keyword  
 c. Both of the above  
 d. None of the above  
  
Ans. No Keyword  
  
**Q28.  Which of the following has been introduced with Java 8 ?**  
  
 a. StringBuffer  
 b. StringBuilder  
 c. StringFilter  
 d. StringJoiner  
  
Ans. StringJoiner