# Java 8 - Overview

JAVA 8 is a major feature release of JAVA programming language development. Its initial version was released on 18 March 2014. With the Java 8 release, Java provided supports for functional programming, new JavaScript engine, new APIs for date time manipulation, new streaming API, etc.

## New Features

* **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.

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.

## Scope

Using lambda expression, you can refer to any 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.

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);

}

}

Java 8 - Method References

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

* Static methods
* Instance methods
* Constructors using new operator (TreeSet::new)

# Java 8 - 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. Following is the list of functional interfaces defined in java.util.Function package.

|  |  |
| --- | --- |
| **S. No.** | **Interface & Description** |
| 1 | **BiConsumer<T,U>**  Represents an operation that accepts two input arguments, and returns no result. |
| 2 | **BiFunction<T,U,R>**  Represents a function that accepts two arguments and produces a result. |
| 3 | **BinaryOperator<T>**  Represents an operation upon two operands of the same type, producing a result of the same type as the operands. |
| 4 | **BiPredicate<T,U>**  Represents a predicate (Boolean-valued function) of two arguments. |
| 5 | **BooleanSupplier**  Represents a supplier of Boolean-valued results. |
| 6 | **Consumer<T>**  Represents an operation that accepts a single input argument and returns no result. |
| 7 | **DoubleBinaryOperator**  Represents an operation upon two double-valued operands and producing a double-valued result. |
| 8 | **DoubleConsumer**  Represents an operation that accepts a single double-valued argument and returns no result. |
| 9 | **DoubleFunction<R>**  Represents a function that accepts a double-valued argument and produces a result. |
| 10 | **DoublePredicate**  Represents a predicate (Boolean-valued function) of one double-valued argument. |
| 11 | **DoubleSupplier**  Represents a supplier of double-valued results. |
| 12 | **DoubleToIntFunction**  Represents a function that accepts a double-valued argument and produces an int-valued result. |
| 13 | **DoubleToLongFunction**  Represents a function that accepts a double-valued argument and produces a long-valued result. |
| 14 | **DoubleUnaryOperator**  Represents an operation on a single double-valued operand that produces a double-valued result. |
| 15 | **Function<T,R>**  Represents a function that accepts one argument and produces a result. |
| 16 | **IntBinaryOperator**  Represents an operation upon two int-valued operands and produces an int-valued result. |
| 17 | **IntConsumer**  Represents an operation that accepts a single int-valued argument and returns no result. |
| 18 | **IntFunction<R>**  Represents a function that accepts an int-valued argument and produces a result. |
| 19 | **IntPredicate**  Represents a predicate (Boolean-valued function) of one int-valued argument. |
| 20 | **IntSupplier**  Represents a supplier of int-valued results. |
| 21 | **IntToDoubleFunction**  Represents a function that accepts an int-valued argument and produces a double-valued result. |
| 22 | **IntToLongFunction**  Represents a function that accepts an int-valued argument and produces a long-valued result. |
| 23 | **IntUnaryOperator**  Represents an operation on a single int-valued operand that produces an int-valued result. |
| 24 | **LongBinaryOperator**  Represents an operation upon two long-valued operands and produces a long-valued result. |
| 25 | **LongConsumer**  Represents an operation that accepts a single long-valued argument and returns no result. |
| 26 | **LongFunction<R>**  Represents a function that accepts a long-valued argument and produces a result. |
| 27 | **LongPredicate**  Represents a predicate (Boolean-valued function) of one long-valued argument. |
| 28 | **LongSupplier**  Represents a supplier of long-valued results. |
| 29 | **LongToDoubleFunction**  Represents a function that accepts a long-valued argument and produces a double-valued result. |
| 30 | **LongToIntFunction**  Represents a function that accepts a long-valued argument and produces an int-valued result. |
| 31 | **LongUnaryOperator**  Represents an operation on a single long-valued operand that produces a long-valued result. |
| 32 | **ObjDoubleConsumer<T>**  Represents an operation that accepts an object-valued and a double-valued argument, and returns no result. |
| 33 | **ObjIntConsumer<T>**  Represents an operation that accepts an object-valued and an int-valued argument, and returns no result. |
| 34 | **ObjLongConsumer<T>**  Represents an operation that accepts an object-valued and a long-valued argument, and returns no result. |
| 35 | **Predicate<T>**  Represents a predicate (Boolean-valued function) of one argument. |
| 36 | **Supplier<T>**  Represents a supplier of results. |
| 37 | **ToDoubleBiFunction<T,U>**  Represents a function that accepts two arguments and produces a double-valued result. |
| 38 | **ToDoubleFunction<T>**  Represents a function that produces a double-valued result. |
| 39 | **ToIntBiFunction<T,U>**  Represents a function that accepts two arguments and produces an int-valued result. |
| 40 | **ToIntFunction<T>**  Represents a function that produces an int-valued result. |
| 41 | **ToLongBiFunction<T,U>**  Represents a function that accepts two arguments and produces a long-valued result. |
| 42 | **ToLongFunction<T>**  Represents a function that produces a long-valued result. |
| 43 | **UnaryOperator<T>**  Represents an operation on a single operand that produces a result of the same type as its operand. |

## Functional Interface Example

Predicate <T> interface is a functional interface with a method test(Object) to return a Boolean value. This interface signifies that an object is tested to be true or false.

To get more clarity on this, write the following program in an code editor and verify the results.

# Java8 - 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!!!");

}

}

# 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());

# Java 8 - Optional Class

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.

## Class Declaration

Following is the declaration for **java.util.Optional<T>** class −

public final class Optional<T>

extends Object

## Class Method

|  |  |
| --- | --- |
| **S. No.** | **Method & Description** |
| 1 | **static <T> Optional<T> empty()**  Returns an empty Optional instance. |
| 2 | **boolean equals(Object obj)**  Indicates whether some other object is "equal to" this Optional. |
| 3 | **Optional<T> filter(Predicate<? super <T> predicate)**  If a value is present and the value matches a given predicate, it returns an Optional describing the value, otherwise returns an empty Optional. |
| 4 | **<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)**  If a value is present, it applies the provided Optional-bearing mapping function to it, returns that result, otherwise returns an empty Optional. |
| 5 | **T get()**  If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException. |
| 6 | **int hashCode()**  Returns the hash code value of the present value, if any, or 0 (zero) if no value is present. |
| 7 | **void ifPresent(Consumer<? super T> consumer)**  If a value is present, it invokes the specified consumer with the value, otherwise does nothing. |
| 8 | **boolean isPresent()**  Returns true if there is a value present, otherwise false. |
| 9 | **<U>Optional<U> map(Function<? super T,? extends U> mapper)**  If a value is present, applies the provided mapping function to it, and if the result is non-null, returns an Optional describing the result. |
| 10 | **static <T> Optional<T> of(T value)**  Returns an Optional with the specified present non-null value. |
| 11 | **static <T> Optional<T> ofNullable(T value)**  Returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional. |
| 12 | **T orElse(T other)**  Returns the value if present, otherwise returns other. |
| 13 | **T orElseGet(Supplier<? extends T> other)**  Returns the value if present, otherwise invokes other and returns the result of that invocation. |
| 14 | **<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)**  Returns the contained value, if present, otherwise throws an exception to be created by the provided supplier. |
| 15 | **String toString()**  Returns a non-empty string representation of this Optional suitable for debugging. |

**Note** − This class inherits methods from the **java.lang.Object** class.

# 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.

# Java 8 - New Date/Time API

With Java 8, a new Date-Time API is introduced to cover the following drawbacks of old date-time API −

* **Not thread safe** − java.util.Date is not thread safe, thus developers have to deal with concurrency issue while using date. The new date-time API is immutable and does not have setter methods.
* **Poor design** − Default Date starts from 1900, month starts from 1, and day starts from 0, so no uniformity. The old API had less direct methods for date operations. The new API provides numerous utility methods for such operations.
* **Difficult time zone handling** − Developers had to write a lot of code to deal with timezone issues. The new API has been developed keeping domain-specific design in mind.

Java 8 introduces a new date-time API under the package **java.time**. Following are some of the important classes introduced in java.time package −

* **Local** − Simplified date-time API with no complexity of timezone handling.
* **Zoned** − Specialized date-time API to deal with various timezones.

## Local Data-Time API

LocalDate/LocalTime and LocalDateTime classes simplify the development where timezones are not required.

## Zoned Date-Time API

Zoned date-time API is to be used when time zone is to be considered

## Chrono Units Enum

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

## Period & Duration

With Java 8, two specialized classes are introduced to deal with the time differences −

* **Period** − It deals with date based amount of time.
* **Duration** − It deals with time based amount of time.

## Temporal Adjusters

TemporalAdjuster is used to perform the date mathematics. For example, get the "Second Saturday of the Month" or "Next Tuesday".

## Backward Compatibility

A **toInstant()** method is added to the original Date and Calendar objects, which can be used to convert them to the new Date-Time API. Use an ofInstant(Insant,ZoneId) method to get a LocalDateTime or ZonedDateTime object.

# 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.