# Java 8 Features with Examples:

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

1. **[Functional interface -](#_1._Functional_interface:)** Used in Lamda expression.
2. **[Default method](#_2._Default_method:)**− Interface to have default method implementation.

**3. [Lambda expression](#_3._Lambda_expression:)**− Adds functional processing capability to Java.

**4. [Method references −](#_Functional_interface:)** Referencing functions by their names instead of invoking them directly. Using functions as parameter.

**5. [Java forEach loop](#_Java_forEach_loop) -** provides a new method forEach() to iterate the elements.

**6. [StringJoiner](#_StringJoiner_Class:) Class -** It is used to construct a sequence of characters separated by a delimiter.

**[7. Java Optional Class](#_7._Java_Optional) -** It is a public final class and used to deal with NullPointerException in Java application.

**[8. Java Parallel Array Sorting](#_8._Java_Parallel)**:Used to sort array elements.

**[9. Java Nashorn:](#_9._Java_Nashorn:)** for java script engine.

[10. Streams](#_10._Streams): Using stream, you can process data in a declarative way similar to SQL statements.

* **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.
* **PermGen Space Removed**

The **PermGen** space is removed from Java 8 and instead we have **MetaSpace** introduced. One of the most dreaded error, “java.lang.OutOfMemoryError: PermGen error” will no longer be seen from Java 8. Nice thing is that the MetaSpace default is unlimited and that the system **memory itself becomes the memory.**

* **Elvis operator.**

# 1. Functional interface:

* Functional interfaces have a single functionality to exhibit. For example, a Comparable interface with a single method ‘compareTo’ is used for comparison purpose.
* An **interface with exactly one abstract method is called Functional Interface**. @FunctionalInterface annotation is added so that we can mark an interface as functional interface.
* It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.
* The major benefit of java 8 functional interfaces is that we can use **lambda expressions** to instantiate them and avoid using bulky anonymous class implementation.
* The interface can also declare the abstract methods from the java.lang.Object class, but still the interface can be called as a Functional Interface:

**Example:** java.lang.Runnable is a great example of functional interface with single abstract method run().

Functional Interface is also known as **Single Abstract Method Interfaces** or **SAM Interfaces**. It is a new feature in Java, which helps to achieve functional programming approach.

**package** com.capgemini;

@FunctionalInterface

**interface** functionalInterfaceDdd {

**void** addNum(**int** x); // abstract method in java8, with no body.

**default** **int** subNum(**int** a, **int** b) {

**return** a - b;

}

}

**class** FunctionalInterfaceExa **implements** functionalInterfaceDdd {

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

FunctionalInterfaceExa in = **new** FunctionalInterfaceExa();

in.addNum(12);

System.***out***.println(in.subNum(12, 10)); // This is optional.

}

@Override

**public** **void** addNum(**int** x) {

x = x \* 5;

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

}}

**O/P:**

Hello there

A functional interface can have methods of object class. See in the following example.

**package** com.capgemini.test;

@FunctionalInterface

**interface** Addition{

**void** say(String msg); // abstract method

// It can contain any number of Object class methods.

**int** hashCode();

String toString();

**boolean** equals(Object obj);

}

**public** **class** FunctionalInterfaceExample2 **implements** sayable{

**public** **void** say(String msg){

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

}

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

FunctionalInterfaceExample2 fie = **new** FunctionalInterfaceExample2();

fie.say("Hello there");

}

}

**O/P:**

Hello there.

**Invalid functional interface:**

**interface** sayable{

**void** say(String msg); // abstract method

}

@FunctionalInterface

**interface** Doable **extends** sayable{

// Invalid '@FunctionalInterface' annotation; Doable is not a functional interface

**void** doIt();

}

Output:

compile-time error

# 2. Default method:

Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default are known as default methods. These methods are **non-abstract** methods.

**interface** i1 {

**static** **void** m1() {

System.***out***.println("i1- m1()");

}

**default** **void** m2() {

System.***out***.println("i1- m2()");

}

}

**public** **class** DefaultMethod **implements** i1 {

**public** **void** m3(String name) {

System.***out***.println("m3 : " + name);

}

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

DefaultMethod defaultMethod = **new** DefaultMethod();

defaultMethod.m3("premchand");

defaultMethod.m2();

i1.*m1*();

}

}

**O/P:**

m3 : premchand

i1- m2()

i1- m1()

**Static Methods inside Java 8 Interface**

You can also define static methods inside the interface. Static methods are used to define utility methods. The following example explain, how to implement static method in interface?

**package** com.capgemini.test;

**interface** Sayable{

// default method

**default** **void** say(){

System.***out***.println("Hello, this is default method");

}

// Abstract method

**void** sayMore(String msg);

// static method

**static** **void** sayLouder(String msg){

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

}

}

**public** **class** DefaultMethods **implements** Sayable{

**public** **void** sayMore(String msg){ // implementing abstract method

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

}

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

DefaultMethods dm = **new** DefaultMethods();

dm.say(); // calling default method

dm.sayMore("Work is worship"); // calling abstract method

Sayable.*sayLouder*("Helloooo..."); // calling static method

}

}

**O/P:**

Hello, this is default method

Work is worship

Helloooo...

# 3. Lambda expression:

Lambda expression facilitates functional programming, and simplifies the development a lot.

**Syntax:**

A lambda expression is characterized by the following syntax.

parameter -> expression body

**package** com.capgemini.test;

**class** Lambda\_Expressions {

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

Lambda\_Expressions tester = **new** Lambda\_Expressions();

// with type declaration

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

// without 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));

// without parenthesis

GreetingService greetService1 = message -> System.***out***.println("Hello " + message);

// with 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);

}

}

**O/P:**

10 + 5 = 15

10 - 5 = 5

10 x 5 = 50

10 / 5 = 2

Hello Mahesh

Hello Suresh

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.

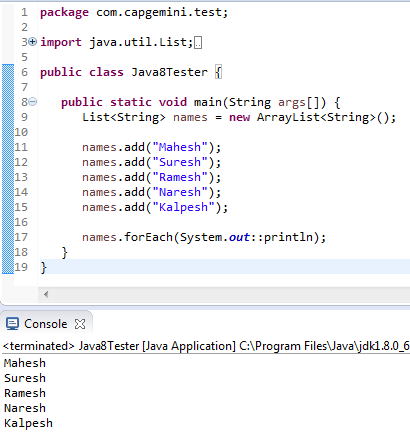
***Conclusion:***

* Lambda expressions are used primarily to define inline implementation of a **functional interface**, i.e., an interface with a single method only.
* Lambda expression eliminates the need of anonymous class and gives a very simple yet powerful functional programming capability to Java.

# 4. Method references

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

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.



* **Reference to a Static Method**

**package** com.capgemini.test;

**interface** Sayable {

**void** say();

}

**public** **class** MethodReference {

**public** **static** **void** saySomething() {

System.***out***.println("Hello, this is static method.");

}

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

// Referring static method

Sayable sayable = MethodReference::*saySomething*;

// Calling interface method

sayable.say();

}

}

**O/P:**

Hello, this is static method.

* **Reference to an Instance Method**

**package** com.capgemini.test;

**interface** Sayable{

**void** say();

}

**public** **class** MethodReference {

**public** **void** saySomething(){

System.***out***.println("Hello, this is non-static method.");

}

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

MethodReference methodReference = **new** MethodReference(); // Creating object

// Referring non-static method using reference

Sayable sayable = methodReference::saySomething;

// Calling interface method

sayable.say();

// Referring non-static method using anonymous object

Sayable sayable2 = **new** MethodReference()::saySomething;

// You can use anonymous object also

// Calling interface method

sayable2.say();

}

}

**O/P:**

Hello, this is non-static method.

Hello, this is non-static method.

* **Reference to a Constructor**

**package** com.capgemini.test;

**interface** Messageable {

Message getMessage(String msg);

}

**class** Message {

Message(String msg) {

System.***out***.print(msg);

}

}

**public** **class** ConstructorReference {

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

Messageable hello = Message::**new**;

hello.getMessage("Hello Premchand");

}

}

**O/P:**

Hello Premchand

# 5. Java forEach loop

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

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

**package** com.capgemini.test;

**import** java.util.ArrayList;

**import** java.util.List;

**class** ForEachLoop {

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

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

items.add("A");

items.add("B");

items.add("C");

items.add("D");

items.add("E");

// lambda

// Output : A,B,C,D,E

items.forEach(item -> System.***out***.println(item));

// Output : C

items.forEach(item -> {

**if** ("C".equals(item)) {

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

}

});

// method reference

// Output : A,B,C,D,E

items.forEach(System.***out***::println);

// Stream and filter

// Output : B

items.stream().filter(s -> s.contains("B")).forEach(System.***out***::println);

}

}

# 6. StringJoiner Class:

Java added a new final class StringJoiner in java.util package. It is used to construct a sequence of characters separated by a delimiter. Now, you can create string by passing delimiters like comma(,), hyphen(-) etc. You can also pass prefix and suffix to the char sequence.

**package** com.capgemini.test;

**import** java.util.StringJoiner;

**public** **class** StringJoinerExample {

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

// StringJoiner joinNames = new StringJoiner("," , "[","]"); // passing comma(,) as delimiter

StringJoiner joinNames = **new** StringJoiner("," , "[","]"); // passing comma(,) as delimiter

// Adding values to StringJoiner

joinNames.add("Rahul");

joinNames.add("Raju");

joinNames.add("Peter");

joinNames.add("Raheem");

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

}

}

O/P:

[Rahul,Raju,Peter,Raheem]

# 7. Java Optional Class

Java introduced a new class Optional in jdk8. It is a public final class and used to deal with NullPointerException in Java application. You must import java.util package to use this class. It provides methods which are used to check the presence of value for particular variable.

**package** com.capgemini.test;

**import** java.util.Optional;

**public** **class** OptionalExample {

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

String[] str = **new** String[10];

Optional<String> checkNull = Optional.*ofNullable*(str[5]);

**if** (checkNull.isPresent()) { // check for value is present or not

String lowercaseString = str[5].toLowerCase();

System.***out***.print(lowercaseString);

} **else**

System.***out***.println("string value is not present");

}

}

O/P:

string value is not present

# 

# 8. Java Parallel Array Sorting

Java provides a new additional feature in Array class which is used to sort array elements parallel.New methods has added to java.util.Arrays package.

**package** com.capgemini.test;

**import** java.util.Arrays;

**public** **class** ParallelArraySorting {

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

**int**[] arr = { 5, 8, 1, 0, 6, 9 };

**for** (**int** i : arr) {

System.***out***.print(i + " ");

}

Arrays.*parallelSort*(arr);

System.***out***.println("\nArray elements after sorting");

**for** (**int** i : arr) {

System.***out***.print(i + " ");

}

}

}

**O/P:**

5 8 1 0 6 9

Array elements after sorting

0 1 5 6 8 9

# 9. Java Nashorn:

Nashorn is a JavaScript engine. It is used to execute JavaScript code dynamically at JVM (Java Virtual Machine). Java provides a command-line tool jjs which is used to execute JavaScript code.

**package** com.capgemini.test;

**import** javax.script.\*;

**import** java.io.\*;

**public** **class** NashornExample {

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

// Creating script engine

ScriptEngine ee = **new** ScriptEngineManager().getEngineByName("Nashorn");

// Reading Nashorn file

ee.eval(**new** FileReader("hello.js"));

}

}

hello.js :

var hello = function(){

print("welcome to Hello Nashorn");

};

hello();

**O/P:**

Welcome to Hello Nashorn

# 10. 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.

*The* ***source*** *here refers to a Collection, IO Operation or Arrays who provides data to a Stream.* ***Stream*** *keeps the order of the data as it is in the source. Just like functional programming languages, Streams support Aggregate Operations.*

*The* ***common aggregate operations*** *are filter, map, reduce, find, match, and sort. These operations can be executed in series or in parallel.*

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.

**Stream API’s:**

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

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

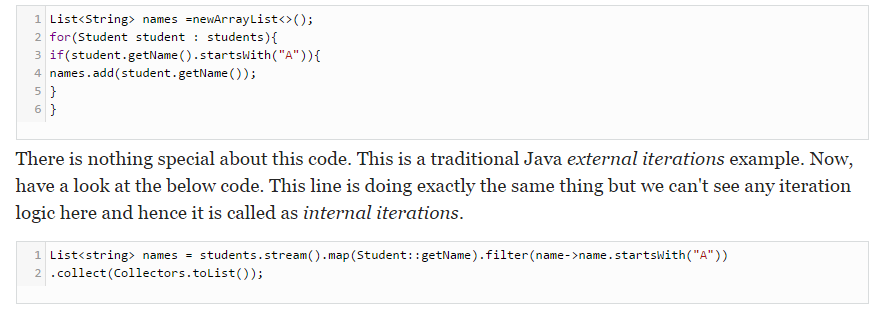
productsList.add(**new** Product(5,"Apple Laptop",90000f));

**double** totalPrice3 = productsList.stream()

                        .collect(**Collectors.summingDouble**(product->product.price));

System.out.println(totalPrice3);

**Example-2.**

****

**Kinds of stream:**

Sequential streams & parallelStream()