# Java 8 Concepts

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.

Lambda Expression

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

parameter -> expression body

* 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.

//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;

//without parenthesis

GreetingService greetService1 = message ->

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

* 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.

effectively final variable: A variable which is **not declared as final** but whose value is never changed after initialization is effectively final.

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 −
  + Static methods
  + Instance methods
  + Constructors using new operator (TreeSet::new)

public static void main(String args[]) {

List names = new ArrayList();

names.add("Mahesh");

names.add("Suresh");

names.add("Ramesh");

names.add("Naresh");

names.add("Kalpesh");

names.forEach(System.out::println);

}

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.

Default Methods

* 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.

public interface vehicle {

default void print() {

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

}

static void blowHorn() {

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

}

}

Stream API

* 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 represents a sequence of objects from a source, which supports aggregate operations
* Collection interface has two methods to generate a Stream.
  + **stream()**
  + **parallelStream()**

## forEach

Random random = new Random();

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

## map

//get list of unique squares

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

## fliter

//get count of empty string

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

## limit

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

## sorted

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

## Parallel Processing

//get count of empty string

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

## Collectors

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

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

Optional Class

* Optional is a container object 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.

public final class Optional<T> extends Object

Example

//Optional.ofNullable - allows passed parameter to be null.

Optional<Integer> a = Optional.ofNullable(value1);

//Optional.of - throws NullPointerException if passed parameter is null

Optional<Integer> b = Optional.of(value2);

//Optional.isPresent - checks the value is present or not

System.out.println("First parameter is present: " + a.isPresent());

//Optional.orElse - returns the value if present otherwise returns

//the default value passed.

Integer value1 = a.orElse(new Integer(0));

//Optional.get - gets the value, value should be present

Integer value2 = b.get();

Interview Questions