=================

Java 1.8v Features

=================

-> Java 1.8v introduced lot of new features in java

-> Java 1.8v new features changed java programming style

=========================

Main Objectivies of Java 1.8v

=========================

-> Simplify Java Programming

-> Enable Functional Programming

-> Write more readable and consice code

=================

Java 1.8 Features

=================

1) Interface changes

1.1 ) Default Methods

1.2 ) Static Methods

2) Functional Interfaces (@FunctionalInterface)

2.1 ) Predicate

2.2 ) Consumer

2.3 ) Supplier

2.4 ) Function

3) Lambda Expressions

5) \*\*\*\*\*\* Stream API \*\*\*\*\*\*\*\*

6) Optional class (to avoid null pointer exceptions)

7) StringJoiner

8) forEach ( ) method

9) Date & Time API

================

Interface changes

================

-> Interface means collection of abstract methods

Note: The method which doesn't contain body is called as abstract method

-> A class can implement interface using "implements"

-> When a class is implementing interface its mandatory that class should implement all abstract methods of that interface othewise class can't be compile.

=> Here i am taking one interface with one abstract method. All the classes which are implementing that interface should overide interface method(s).

interface Vehicle {

public abstract void startVechicle ( );

}

class Car implements Vehicle {

public void startVehicle ( ) {

// logic to start car

}

}

class Bus implements Vehicle {

public void startVehicle ( ) {

// logic to start bus

}

}

class Bike implements Vehicle {

public void startVehicle ( ) {

// logic to start bike

}

}

=> If we add new method in interface then Car, Bike and Bus will fail at compile time.

=> To overcome above problem we will use Default & Static methods

1) Interface can have concreate methods from 1.8v

2) Interface concrete method should be default or static

3) interface default methods we can override in impl classes

4) interface static methods we can't overide in impl classes

5) We can write multiple default & static methods in interface

6) Default & Static method introduced to provide backward compatability

Ex: forEach ( ) method added in java.util.Iterable interface as default method in 1.8v

===========

package in.com;

interface Vehicle {

public void start();

public default void m1() {

}

public default void m2() {

}

public static void clean() {

System.out.println("cleaning completed...");

}

}

public class Car implements Vehicle {

public void start() {

System.out.println("car started...");

}

public static void main(String[] args) {

Car c = new Car();

Vehicle.clean();

c.start();

}

}

============================================

=====================

Lambda Expressions

====================

-> Introdced in java 1.8v

-> Java is called as Object Oriented Programming language. Everything will be represented using Classes and Objects.

-> From 1.8v onwards Java is also called as Functional Programming Language.

-> In OOP language Classes & Objects are main entities. We need to write methods inside the class only.

-> Functional Programming means everything will be represented in the form functions. Functions can exist outside of the class. Functions can be stored into a reference variable. A function can be passed as a parameter to other methods.

-> Lambda Expressions introduced in Java to enable Functional Programming.

==============

What is Lambda

==============

-> Lambda is an anonymous function

- No Name

- No Modifier

- No Return Type

Ex:-1

public void m1 ( ) {

s.o.p("hi");

}

( ) -> { s.o.p ("hi") }

Note: When we have single line in body then curly braces are optional

( ) -> s.o.p ("hi");

Ex:-2

public void add (int a, int b){

s.o.p(a+b);

}

( int a, int b) -> { s.o.p (a+b) } ;

(or)

(int a, int b) -> s.o.p (a+b);

(or)

Lambda Expression : (a, b) -> s.o.p(a+b);

Ex:-3

public int getLength (String name) {

return name.length ( );

}

(String name) -> { return name.length ( ) };

(String name) -> return name.length ( ) ;

(name) -> return name.length ( );

Lambda Expression : name -> name.length ( ) ;

Ex:-4

public Double getEmpSalary (Employee emp) {

return emp.getSalary ( );

}

Lambda Expression : emp -> emp.getSalary ( );

==================

Functional Interfaces

==================

-> The interface which contains only one abstract method is called as Functional Interface

-> Functional Interfaces are used to invoke Lambda expressions

-> Below are some predefined functional interfaces

Runnable ------------> run ( ) method

Callable ----------> call ( ) method

Comparable -------> compareTo ( )

-> To represent one interface as Functional Interface we will use @FunctionalInterface annotation.

@FunctionalInterface

public interface MyInterface {

public void m1( );

}

Note: When we write @FunctionalInterface then our compiler will check interface contains only one abstract method or not.

-> In Java 8 several predefined Functional interfaces got introduced they are

1) Predicate

2) Consumer

3) Supplier

4) Function

-> The above interfaces are provided in java.util.function package

========

Predicate

========

-> It is predefined Functional interface

-> It is used check condition and returns true or false value

-> Predicate interface having only one abstract method that is test (T t)

interface Predicate{

boolean test(T t);

}

// Predicate Example

package in.com.java8;

import java.util.function.Predicate;

public class PredicateDemo {

public static void main(String[] args) {

Predicate<Integer> p = i -> i > 10;

System.out.println(p.test(5));

System.out.println(p.test(15));

}

}

=======================================================================================

Task: Declare names in an array and print names which are starting with 'A' using lambda expression.

String[ ] names = {"Anushka", "Anupama", "Deepika", "Kajol", "Sunny" };

=========================================================================================

package in.com.java8;

import java.util.function.Predicate;

public class PredicateDemo2 {

public static void main(String[] args) {

String[ ] names = { "Anushka", "Anupama", "Deepika", "Kajol", "Sunny" };

Predicate<String> p = name -> name.charAt(0) == 'A';

for (String name : names) {

if ( p.test(name) ) {

System.out.println(name);

}

}

}

}

============================================================================

Task-2 : Take list of persons and print persons whose age is >= 18 using Lambda Expression

============================================================================

package in.com.java8;

import java.util.Arrays;

import java.util.List;

import java.util.function.Predicate;

class Person {

String name;

int age;

Person(String name, int age) {

this.name = name;

this.age = age;

}

}

public class PredicatePersonsDemo {

public static void main(String[] args) {

Person p1 = new Person("John", 26);

Person p2 = new Person("Smith", 16);

Person p3 = new Person("Raja", 36);

Person p4 = new Person("Rani", 6);

List<Person> persons = Arrays.asList(p1, p2, p3, p4);

Predicate<Person> predicate = p -> p.age >= 18;

for (Person person : persons) {

if (predicate.test(person)) {

System.out.println(person.name);

}

}

}

}

==========================

Supplier Functional Interface

==========================

-> Supplier is a predefined functional interface introduced in java 1.8v

-> It contains only one abstract method that is get ( ) method

-> Supplier interface will not take any input, it will only returns the value.

Ex:

----

OTP Generation

package in.com.java8;

import java.util.function.Supplier;

public class SupplierDemo {

public static void main(String[] args) {

Supplier<String> s = () -> {

String otp = "";

for (int i = 1; i <= 6; i++) {

otp = otp + (int) (Math.random() \* 10);

}

return otp;

};

System.out.println(s.get());

System.out.println(s.get());

System.out.println(s.get());

System.out.println(s.get());

System.out.println(s.get());

System.out.println(s.get());

}

}

==========================

Consumer Functional Interface

==========================

-> Consumer is predefined functional interface

-> It contains one abstract method i.e accept (T t)

-> Consumer will accept input but it won't return anything

Note: in java 8 forEach ( ) method got introduced. forEach(Consumer consumer) method will take Consumer as parameter.

package in.com.java8;

import java.util.Arrays;

import java.util.List;

import java.util.function.Consumer;

public class ConsumerDemo {

public static void main(String[] args) {

Consumer<String> c = (name) -> System.out.println(name + ", Good Evening");

c.accept("Alex");

c.accept("John");

c.accept("Rani");

List<Integer> numbers = Arrays.asList(10, 20, 30, 40);

// for loop

// for each loop

// iterator

// list iterator

numbers.forEach(i -> System.out.println(i));

}

}

=========================================================

Retrieve student record based on student id and return that record

=========================================================

Predicate ------> takes inputs ----> returns true or false ===> test ( )

Supplier -----> will not take any input---> returns output ===> get ( )

Consumer ----> will take input ----> will not return anything ===> accept ( )

Function -----> will take input ---> will return output ===> apply ( )

=========================

Function Functional Interface

=========================

-> Function is predefined functional interface

-> Funcation interface having one abstract method i.e apply(T r)

interface Function<R,T>{

R apply (T t);

}

-> It takes input and it returns output

package in.com.java8;

import java.util.function.Function;

public class FunctionDemo {

public static void main(String[] args) {

Function<String, Integer> f = (name) -> name.length();

System.out.println(f.apply("gaurav"));

System.out.println(f.apply("pune"));

System.out.println(f.apply("sachin"));

}

}

=========================================================

Task : Take 2 inputs and perform sum of two inputs and return ouput

=========================================================

BiFunction<Integer,Integer,Integer> bif = (a,b) -> a+b;

Integer sum = bi.apply(10,20);

==========================

forEach (Consumer c) method

===========================

-> forEach (Consumer c) method introduced in java 1.8v

-> forEach ( ) method added in Iterable interface

-> forEach ( ) method is a default method (it is having body)

-> This is method is used to access each element of the collection (traverse collection from start to end)

package in.com.java8;

import java.util.ArrayList;

public class NumbersSort1 {

public static void main(String[] args) {

ArrayList<Integer> al = new ArrayList<>();

al.add(5);

al.add(3);

al.add(4);

al.add(1);

al.add(2);

al.forEach(i -> System.out.println(i));

}

}

==============

StringJoiner

==============

-> java.util.StringJoiner class introduced in java 1.8v

-> It is used to join more than one String with specified delimiter

-> We can concat prefix and suffix while joininging strings using StringJoiner

StringJoiner sj = new StringJoiner (CharSequence delim);

package in.com.java8;

import java.util.StringJoiner;

public class StringJoinerDemo {

public static void main(String[] args) {

StringJoiner sj1 = new StringJoiner("-");

sj1.add("Gaurav");

sj1.add("Solapure");

System.out.println(sj1); // Gaurav-solapure

}

}

=============

Optional Class

=============

-> java.util.Optional class introduced in java 1.8v

-> Optional class is used to avoid NullPointerExceptions in the program

Q) What is NullPointerException (NPE) ?

Ans) When we perform some operation on null value then we will get NullPointerException

String s = null;

s.length ( ) ; // NPE

-> To avoid NullPointerExceptions we have to implement null check before performing operation on the Object like below.

String s = null;

if( s! = null ) {

System.out.println(s.length ( ));

}

Note: In project there is no gaurantee that every programmer will implement null checks. If any body forgot to implement null check then program will run into NullPointerException.

-> To avoid this problem we need to use Optional class like below.

package in.com.java8;

import java.util.Optional;

public class User {

// Without Optional object

public String getUsernameById(Integer id) {

if (id == 100) {

return "Raju";

} else if (id == 101) {

return "Rani";

} else if (id == 102) {

return "John";

} else {

return null;

}

}

// with Optional Object

public Optional<String> getUsername(Integer id) {

String name = null;

if (id == 100) {

name = "Raju";

} else if (id == 101) {

name = "Rani";

} else if (id == 102) {

name = "John";

}

return Optional.ofNullable(name);

}

}

package in.com.java8;

import java.util.Optional;

import java.util.Scanner;

public class MsgService {

public static void main(String[] args) {

Scanner s = new Scanner(System.in);

System.out.println("Enter User ID");

int userId = s.nextInt();

User u = new User();

/\*String userName = u.getUsernameById(userId);

String msg = userName.toUpperCase() + ", Hello";

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

Optional<String> username = u.getUsername(userId);

if(username.isPresent()) {

String name = username.get();

System.out.println(name.toUpperCase()+", Hello");

}else {

System.out.println("No Data Found");

}

}

}

=======================

Date & Time API Changes

=======================

-> In java we have below 2 classes to represent Date

1) java.util.Date

2) java.sql.Date

=> In java 1.8 version, new classes got introduced to deal with Date & Time functionalities

1) java.time.LocalDate (it will deal with only date)

2) java.time.LocalTime (it will deal with only time)

3) java.time.LocalDateTime (it will deal with both date & time)

// Java 1.8 Date API Example

package in.com.java8;

import java.time.Duration;

import java.time.LocalDate;

import java.time.LocalDateTime;

import java.time.LocalTime;

import java.time.Period;

public class NewDateDemo {

public static void main(String[] args) {

LocalDate of = LocalDate.of(2021, 1, 20);

System.out.println(of);

LocalDate date = LocalDate.now();

System.out.println(date);

date = date.plusDays(3);

System.out.println(date);

date = date.plusMonths(1);

System.out.println(date);

date = date.plusYears(2);

System.out.println(date);

boolean leapYear = LocalDate.parse("2020-12-22").isLeapYear();

System.out.println("Leap Year :: " + leapYear);

boolean before = LocalDate.parse("2021-12-22").isBefore(LocalDate.parse("2022-12-22"));

System.out.println("Before Date : " + before);

LocalTime time = LocalTime.now();

System.out.println(time);

time = time.plusHours(2);

System.out.println(time);

LocalDateTime datetime = LocalDateTime.now();

System.out.println(datetime);

Period period = Period.between(LocalDate.parse("1991-05-20"), LocalDate.now());

System.out.println(period);

Duration duration = Duration.between(LocalTime.parse("18:00"), LocalTime.now());

System.out.println(duration);

}

}

===========

Stream API

===========

-> Stream API introduced in java 1.8v

-> Stream API is used to process the data

Note: Collections are used to store the data

-> Stream API is one of the major features added in java 1.8v

-> Stream in java can be defined as sequence of elements that comes from a source.

-> Source of data for the Stream can be array or collection

package in.com.streams;

import java.util.Arrays;

import java.util.List;

public class FirstDemo {

public static void main(String[] args) {

List<Integer> list = Arrays.asList(66, 32, 45, 12, 20);

/\*for (Integer i : list) {

if (i > 20) {

System.out.println(i);

}

}\*/

/\*Stream<Integer> stream = list.stream();

Stream<Integer> filteredStrem = stream.filter(i -> i > 20);

filteredStrem.forEach(i -> System.out.println(i));\*/

list.stream().filter(i -> i > 20).forEach(i -> System.out.println(i));

}

}

package in.com.streams;

import java.util.Arrays;

import java.util.List;

public class FirstDemo {

public static void main(String[] args) {

List<String> names = Arrays.asList("John", "Anushka", "David", "Smith", "Alex");

names.stream().filter(i -> i.startsWith("A")).forEach(i -> System.out.println(i));

}

}

=======================

Example-1 : map ( ) method

=======================

public class FirstDemo {

public static void main(String[] args) {

List<String> names = Arrays.asList("india","usa","uk", "japan");

/\*for(String name : names) {

System.out.println(name.toUpperCase());

}\*/

names.stream().map(name -> name.toUpperCase()).forEach(n -> System.out.println(n));

names.stream().mapToInt(name -> name.length()).forEach(i -> System.out.println(i));

}

}