1> What does it mean by Java8 supports Functional Programming?

Before Java 8 everything was mostly object oriented. Except primitives everything in java existed as objects.

All calls to methods/functions are to be made using objects or class references.

Methods/Functions did not exist independently by itself.

With Java 8, functional programming has been introduced. So we can make use of anonymous functions.

Now Java 8 supports Functional programming concepts like lambda expressions, functional interfaces,

--------------------------------------------------------------------------------------------------------------

2> Functional Programming basics -

Functional programming contains the following key concepts:

i> Functions as first class objects

ii> Pure functions

iii> Higher order functions

i> Functions as First Class Objects -

In the functional programming paradigm, functions are first class objects in the language. That means that you can create an "instance" of a function, as have a variable reference that function instance, just like a reference to a String, Map or any other object. Functions can also be passed as parameters to other functions.

ii> Pure functions

A function is a pure function if:

a> The execution of the function has no side effects.

b> The return value of the function depends only on the input parameters passed to the function.

Here is an example of a pure function (method) in Java:

public class ObjectWithPureFunction{

public int sum(int a, int b) {

return a + b;

}

}

Notice how the return value of the sum() function only depends on the input parameters. Notice also that the sum() has no side effects, meaning it does not modify any state (variables) outside the function anywhere.

Contrarily, here is an example of a non-pure function:

public class ObjectWithNonPureFunction{

private int value = 0;

public int add(int nextValue) {

this.value += nextValue;

return this.value;

}

}

Notice how the method add() uses a member variable to calculate its return value, and it also modifies the state of the value member variable, so it has a side effect.

iii> Higher Order Functions

A higher order function is a simple function that can receive function as argument and can return another function as results.

The first example of a higher order function is the Collections.sort() method which takes a Comparator as parameter. Here is an example:

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

list.add("One");

list.add("Abc");

list.add("BCD");

Collections.sort(list, (String a, String b) -> {

return a.compareTo(b);

});

System.out.println(list);

The Collection.sort() takes two parameters. The first parameter is a List and second parameter is a lambda (function). The lambda parameter is what makes Collections.sort() a higher order function

Another example can be -

public class HigherOrderFunctionClass {

public <T> IFactory<T> createFactory(IProducer<T> producer, IConfigurator<T> configurator) {

return () -> {

T instance = producer.produce();

configurator.configure(instance);

return instance;

}

}

}

Notice how the createFactory() method returns a lambda expression as result.

Pure functional programming has a set of rules to follow too:

i> No state

ii> No side effects

iii> Immutable variables

iv> Favour recursion over looping

Rule 1> No State

A rule of the functional programming paradigm is to have no state. By "no state" is typically meant no state external to the function. A function may have local variables containing temporary state internally, but the function cannot reference any member variables of the class or object the function belongs to.

Here is an example of a function that uses no external state:

public class Calculator {

public int sum(int a, int b) {

return a + b;

}

}

Contrarily, here is an example of a function that uses external state:

public class Calculator {

private int initVal = 5;

public int sum(int a) {

return initVal + a;

}

}

This function clearly violates the no state rule.

Rule 2> No Side Effects

This means, that a function cannot change any state outside of the function. Changing state outside of a function including the parameters passed to the function is referred to as a side effect.

State outside of a function refers both to member variables in the class or object the function, and member variables inside parameters to the functions, or state in external systems like file systems or databases

Rule 3> Immutable Variables

A third rule in the functional programming paradigm is that of immutable variables. Immutable variables makes it easier to avoid side effects.

Rule 4> Favour Recursion Over Looping

A fourth rule in the functional programming paradigm is to favour recursion over looping. Recursion uses function calls to achieve looping, so the code becomes more functional.

Another alternative to loops is the Java Streams API. This API is functionally inspired.

--------------------------------------------------------------------------------------------------------------

3> 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:

i> To Utilize Current Multi-Core CPUs Efficiently

Recently, we can observe drastic changes in Hardware. With increase in Database size and growth of multi-code CPU servers, there is need for Java to support such large-scale systems 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.

ii> To Utilize Functional Programming Features in Java

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

iii> To simplifiying programming

To write code more conscisely with less number of line and faster performance

--------------------------------------------------------------------------------------------------------------

4> What are the new features released in Java 8?

The new features released in Java 8 are:

I. Lambda Expression

II. Stream API

III. Date and Time API

IV. Functional Interface

V. Interface Default and Static Methods

VI. Optional

VII. Base64 Encoding and Decoding

VIII. Nashorn JavaScript Engine

IX. Collections API Enhancements

X. Concurrency Enhancements

XI. Fork/Join Framework Enhancements

XII. Spliterator

XIII. Internal Iteration

XIV. Type Annotations and Repeatable Annotations

XV. Method Parameter Reflection

XVI. JVM Parameter Changes

--------------------------------------------------------------------------------------------------------------

5> What are the main benefits of new features introduced in Java 8?

The main benefits of Java 8 features are :

I. Support for functional programming by Lambda and Streams

II. Ease of high volume data processing by Streams

III. Ease of use by getting Parameter names through Reflection

IV. Reusable code with enhanced Collection APIs

V. Smart exception handling with Optional

VI. Control on JVM with new Parameters

VII. Enhanced encryption support with Base 64

VIII. Faster execution with Nashorn JavaScript engine support

--------------------------------------------------------------------------------------------------------------

6> What is Lambda expressions?

Lambda expression is an anonymous function which does not have access modifiers, name or return value declaration.

It accepts a set of input parameters and returns result.

Lambda expressions introduce functional style processing in Java and facilitate the writing of compact and easy-to-read code.

Lambda expression can be passed as a parameter in a method. So we can treat code in Lambda expression as data. This piece of code can be passed to other objects and methods.

--------------------------------------------------------------------------------------------------------------

7> What are the three main parts of a Lambda expression in Java?

Three main parts of a Lambda expression are:

I. Parameter list: A Lambda expression can have zero or more parameters. Parameter list is optional to Lambda.

II. Lambda arrow operator: “->” is known as Lambda arrow operator. It separates the list of parameters and the body of Lambda.

III. Lambda expression body: The piece of code that we want to execute is written in Lambda expression body.

e.g. (Parameter List) ->{expression;}

(s) -> System.out.println(s);

--------------------------------------------------------------------------------------------------------------

8> What are the advantages of a lambda expression?

I. We can pass a lambda expression as an object to a method. This reduces the overhead involved in passing an anonymous class.

We can also pass a method as a parameter to another method using lambda expressions.

II. Less coding

III. We can use lambda expressions if we want a certain action performed on each element of a collection like is streams API

IV. To provide the implementation of the Java 8 Functional Interface.

V. Using Java 8 Lambda expressions higher efficiency can be achieved.

Using CPUs with multicores, user can take advantage of the multicore CPU’s by parallel processing of collections using lambda

--------------------------------------------------------------------------------------------------------------

9> What is MetaSpace in Java8? How does it differ from PermGen Space?

With JDK8, the permGen Space has been removed. So where will the metadata information be stored now?

This metadata is now stored in a native memory are called as "MetaSpace". This memory is not a contiguous Java Heap memory.

It allows for improvements over PermGen space in Garbage collection, auto tuning, concurrent de-allocation of metadata.

--------------------------------------------------------------------------------------------------------------

10> What is a Functional interface in Java 8?

Functional interface in Java is an interface that has exactly one abstract method.

Functional interface is also known as Single Abstract Method Interface(SAM Interface), since it has exactly one abstract method.

It can have multiple default methods with implementation or static methods.

Another way is to mark an Interface with annotation @FunctionalInterface.

Even with the annotation we have to follow the rule of exactly one abstract method.

The only exception to this rule is that if we override java.lang.Object class’s method as an abstract method,

then it does not count as an abstract method.

In Java 8, java.lang.Runnable and java.util.concurrent.Callable are two very popular Functional interfaces.

e.g.

@FunctionalInterface

public interface MyFunctionalInterface {

public void oneMethod(int i, double d); // One abstract method

public String toString(); //valid Object class methods

public boolean equals(Object o); // Valid Object class method

public static int getSum(int a,int b){// valid->method static

return a+b;

}

public default int getMulty(int c,int d){//valid->method default

return c+d;

}

}

Above code compiple properly as it has only Single Abstract Method.

-----------------------------------------------------------------------------------------------------------------------

11> Why do we need Functional interface in Java?

Functional Interfaces are mainly used in Lambda expressions, Method reference and constructor references.

In functional programming, code can be treated as data. For this purpose Lambda expressions are introduced.

They can be used to pass a block of code to another method or object.

Functional Interface serves as a data type for Lambda expressions.

Since a Functional interface contains only one abstract method, the implementation of that method becomes the code that gets passed as an argument to another method

-----------------------------------------------------------------------------------------------------------------------

12> Is it mandatory to use @FunctionalInterface annotation to define a Functional interface in Java 8?

No, it is not mandatory to mark a Functional interface with @FunctionalInterface annotation.

Java does not impose this rule.

But, if we mark an interface with @FunctionalInterface annotation then Java Compiler will give us error in case we define more than one abstract method inside that interface.

e.g.

Invalid '@FunctionalInterface' annotation; Greetings is not a functional interface

-----------------------------------------------------------------------------------------------------------------------

13> Explain the rules to define a functional interface.

I. Only one interface must be defined having only one abstract method

II. More than on abstract methods cannot be defined

III. A user should make use of @FunctionalInterface annotation in the interface definition.

IV. Any number of different methods like the default method, static method, etc. can be defined.

V. We can override java.lang.Object class’s method as an abstract method and this will not be counted as an abstract method.

-----------------------------------------------------------------------------------------------------------------------

14> Why Functional Interfaces in Java 8 have single Abstract Method only?

Writing Lambda Expression meaning that we are implementing the interface that is functional interface.

Since a lambda function can only provide the implementation for 1 method so it is mandatory for the functional interface to have ONLY one abstract method.

Also take a note of it, that an interface which does not have single abstract method but it extends another functional interface then the interface is said to be Functional interface too.

-------------------------------------------------------------------------------------------------------------------------

15> Differences between anonymous inner classes and Lambda expression

---------------------------------------------------------------------------------------------------------------------------

16> Can we provide implementation of a method in a Java Interface?

Before Java 8, it was not allowed to provide implementation of a method in an Interface.

Java 8 has introduced the flexibility of providing implementation of a method in an interface. There are two options for that:

I. Default Method: We can give default implementation of a method.

II. Static Method: We can create a static method in an interface and provide implementation.

---------------------------------------------------------------------------------------------------------------------------

17> What is a default method and when do we use it?

Before Java 8, interface has only public abstract methods avaiable.

From Java 8 version, we can able to add concrete methods as well in Interface that is the methods with body.

Default method are those methods in interface which have body and use default keywords.

We can use a default method to add a new functionality to an interface while maintaining backward compatibility with classes that are already implementing the interface:

e.g.

public interface Vehicle {

public void move();

default void hoot() {

System.out.println("peep!");

}

}

Usually, when a new abstract method is added to an interface, all implementing classes will break until they implement the new abstract method.

In Java 8, this problem has been solved by the use of default method.

---------------------------------------------------------------------------------------------------------------------------

18> Can we have a default method definition in the interface without specifying the keyword "default" ?

No. Compiler complains that its an abstract method and hence shouldn't have the body.

---------------------------------------------------------------------------------------------------------------------------

19> 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"

To resolve the problem, go with the following solution.

interface DefaultMethodInterface {

default public void defaultMethod(){

System.out.println("DefaultMethodInterface");

}

}

interface DefaultMethodInterface2 {

default public void defaultMethod(){

System.out.println("DefaultMethodInterface2");

}

}

public class HelloJava8 implements DefaultMethodInterface,DefaultMethodInterface2 {

public void defaultMethod(){

System.out.println("inside hellojava8");

DefaultMethodInterface2.super.defaultMethod();

}

public static void main(String[] args){

HelloJava8 defMethIn = new HelloJava8();

defMethIn.defaultMethod();

}

}

---------------------------------------------------------------------------------------------------------------------------

20> What is the purpose of a Static method in an Interface in Java 8?

From 1.8 version onwards in addition to default methods we can write static methods also inside interface to define utility functions.

As interface static methods by default not available to the implementation class, overriding concept is not applicable. hence by using implementation class reference we can’t call interface static methods.

We should call interface static methods by using interface name.

interface MethodInterface {

public static void sum(int a, int b) {

System.out.println("The Sum:"+(a+b));

}

}

public class HelloJava implements MethodInterface {

public static void main(String[] args){

HelloJava helloJava = new HelloJava();

helloJava.sum(1,3); // Compile time Error

HelloJava.sum(1,3); // Compile time Error

MethodInterface.sum(1, 3);

}

}

Ans : The Sum: 4

From 1.8 version onwards we can write main() method inside interface and hence we can run interface directly from the command prompt.

Ex: interface Interf {

public static void main(String[] args) {

System.out.println("Interface Main Method");

}

}

Ans : JavaInterf

---------------------------------------------------------------------------------------------------------------------------

21> How does Java 8 solve Diamond problem of Multiple Inheritance?

In Multiple Inheritance if a class extends more than one classes with two different implementation of same method then it causes Diamond problem.

Consider following example to see problem and solution for Diamond problem in Java 8:

public interface BaseInterface{

default void display() { //code goes here }

}

public interface BaseOne extends BaseInterface { }

public interface BaseTwo extends BaseInterface { }

public class ChildClass implements BaseOne, BaseTwo { }

In the above code, class ChildClass gives compile time error. Java Compiler cannot decide which display method should it invoke in ChildClass.

To solve this problem, Java SE 8 has given the following remedy:

public interface BaseInterface{

default void display() { //code goes here }

}

public interface BaseOne extends BaseInterface { }

public interface BaseTwo extends BaseInterface { }

public class ChildClass implements BaseOne, BaseTwo {

default void display(){

BaseOne.super.display();

}

}

The method invocation at BaseOne.super.display(); solves the Diamond problem as it resolves the confusion for compiler.

---------------------------------------------------------------------------------------------------------------------------

22> What are the main differences between an interface with default method and an abstract class in Java 8?

Refer to Durga Document

---------------------------------------------------------------------------------------------------------------------------

22> What is a predicate interface ?

A predicate is a function with a single argument and returns boolean value.

Predicate interface present in Java.util.function package.

It’s a functional interface and it contains only one method i.e., test()

Ex:

interface Predicate<T> {

public boolean test(T t);

}

As predicate is a functional interface and hence it can refers lambda expression.

e.g. Write a predicate to check whether the given integer is greater than 10 or not.

import Java.util.function;

class Test {

public static void main(String[] args) {

predicate<Integer> p = i ->(i>10);

System.out.println(p.test(100)); // Ans. true

System.out.println(p.test(7)); // Ans. false

System.out.println(p.test(true)); //Compile time Error

}

}

It’s possible to join predicates into a single predicate by using the following methods.

- and()

- or()

- negate()

these are exactly same as logical AND ,OR complement operators.

e.g.

class test {

public static void main(string[] args) {

int[] x = {0, 5, 10, 15, 20, 25, 30};

predicate<integer> p1 = i->i>10;

predicate<integer> p2=i -> i%2==0;

System.out.println("The Numbers Greater Than 10:");

m1(p1, x);

System.out.println("The Even Numbers Are:");

m1(p2, x);

System.out.println("The Numbers Not Greater Than 10:");

m1(p1.negate(), x);

System.out.println("The Numbers Greater Than 10 And Even Are:);

m1(p1.and(p2), x);

System.out.println("The Numbers Greater Than 10 OR Even:);

m1(p1.or(p2), x);

}

public static void m1(predicate<integer>p, int[] x) {

for(int x1:x) {

if(p.test(x1))

System.out.println(x1);

}

}

---------------------------------------------------------------------------------------------------------------------------

23> What is Functions in Java 8?

Function<T, R> is an in-built functional interface introduced in Java 8 in the java.util.function package.

The primary purpose for which Function<T, R> has been created is for mapping scenarios i.e when an object of a type is taken as input and it is converted(or mapped) to another type.

Common usage of Function is in streams where-in the map function of a stream accepts an instance of Function to convert the stream of one

type to a stream of another type.

Function interface has been defined with the generic types T & R, where T is the type of the input and R is the output type.

Method apply() is the primary abstract functional method of Function interface.

It takes as input a parameter t of type T and gives an output object of type R.

e.g. Write a function to find length of given input string.

import Java.util.function.\*;

class Test {

public static void main(String[] args) {

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

System.out.println(f.apply("Durga")); // Ans. 5

System.out.println(f.apply("Soft")); // Ans. 4

}

}

---------------------------------------------------------------------------------------------------------------------------

24> What is the difference and similarities between Function and Predicate in java 8?

Difference:

Predicate<T> is single argument function and either it returns true or false. This can be used as the assignment target for a lambda expression or method reference.

Function<T,R> is also single argument function but it returns an Object.Here T denotes type of input to the function and R denotes type of Result.

Similarities:

Both are functional interfaces i.e both contain single abstract method.

Refer to durga page 22

---------------------------------------------------------------------------------------------------------------------------

25> What are Consumer and Supplier Functional interface?

The Consumer and Supplier interfaces are a couple of Functional Interfaces that belong to the new Java 8 java.util.function package.

As the package name states, these interfaces are meant to be used together with the new functional Java 8 features.

I. Supplier :

Suppliers represent a function that accepts no arguments and produce a result of some arbitrary type.

i) Suppliers may reference constructor methods:

Supplier referencing a constructor method

Supplier<User> userSupplier = User::new;

User user = userSupplier.get();

Suppliers may also reference static methods:

ii) Supplier referencing a static method

Supplier<User> userSupplier = UserFactory::produceUser;

User user = userSupplier.get();

class UserFactory {

public static User produceUser() {

return new User();

}

}

iii)Instance methods are also available to be referenced by suppliers:

Supplier referencing an instance method

Supplier<User> userSupplier = this::produceUser;

User user = userSupplier.get();

private User produceUser(){

return new User();

}

II. Consumer :

Consumers represent a function that accepts a single argument of an arbitrary type and produce no result:

Simple consumer

Consumer<User> userConsumer = (u) -> System.out.println("Username: "

+ u.getUsername());

userConsumer.accept(user);

Consumers may also be applied to streams of data in order to execute some given action against every stream element:

Consumer applied to a stream

List<User> userList = ...;

userList.stream().forEach((u) -> System.out.println("Username: " + u.getUsername()));

---------------------------------------------------------------------------------------------------------------------------

26> Difference between Predicate, Supplier and Consumer ?

I. Predicate represents an anonymous function that accepts one argument and produces a result.

II. Supplier represents an anonymous function that accepts no argument and produces a result.

III. Consumer represents an anonymous function that accepts an argument and produces no result.

---------------------------------------------------------------------------------------------------------------------------

27> What is method reference in java 8?

Method reference is used refer method of functional interface. It is nothing but compact way of lambda expression.

You can simply replace lambda expression with method reference.

Each time when a lambda expression does nothing but call an existing method, you can replace your lambda expression with method reference.

e.g.

Predicate predicate1 = (n) -> EvenOddCheck.isEven(n);

Using method references, developers can write the above lambda expression as follows.

Predicate predicate2 = EvenOddCheck::isEven;

Double-colon operator i.e. (::) is used for method references.

Developers cannot pass arguments to the method reference. For e.g., they cannot use the method reference for the following lambda expression.

IsReferable demo = () -> ReferenceDemo.commonMethod("Argument in method.");

---------------------------------------------------------------------------------------------------------------------------

28> Explain different types of Method Referance.

There re 4 types of Method Refarance.

I. Reference to a static method.

You can refer to static method defined in the class. Following is the syntax and example which describe the process of referring static method in Java.

Syntax :

ContainingClass::staticMethodName

e.g. 1) In the following example, we have defined a functional interface and referring a static method to it's functional method say().

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();

}

}

Output : Hello, this is static method.

2) interface Predicate {

boolean test(int n);

}

class EvenOddCheck {

public static boolean isEven(int n) {

return n % 2 == 0;

}

}

/\*\*\*\*\* Reference To A Static Method \*\*\*\*\*/

public class MethodReferenceEx1 {

public static void main(String[] args) {

/\*\*\*\* Using Lambda Expression \*\*\*\*/

System.out.println("--------------------Using Lambda Expression----------------------");

Predicate predicate1 = (n) -> EvenOddCheck.isEven(n);

System.out.println(predicate1.test(20));

/\*\*\*\* Using Method Reference \*\*\*\*/

System.out.println("\n---------------------Using Method Reference---------------------");

Predicate predicate2 = EvenOddCheck::isEven;

System.out.println(predicate2.test(25));

}

}

II. Reference to an Instance Method

Syntax

containingObject::instanceMethodName

interface Sayable{

void say();

}

public class InstanceMethodReference {

public void saySomething(){

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

}

public static void main(String[] args) {

InstanceMethodReference methodReference = new InstanceMethodReference(); // 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 InstanceMethodReference()::saySomething; // You can use anonymous object also

// Calling interface method

sayable2.say();

}

}

III. Reference to an Instance Method of an Arbitrary Object of a Particular Type

Syntax :

Class::instanceMethodName

e.g.

public class MethodReferenceEx3 {

public static void main(String[] args) {

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

weeks.add("Monday");

weeks.add("Tuesday");

weeks.add("Wednesday");

/\*\*\*\* Using Lambda Expression \*\*\*\*/

System.out.println("--------------------Using Lambda Expression----------------------");

weeks.stream().map((s)-> s.toUpperCase()).forEach((s)->System.out.println(s));

/\*\*\*\* Using Method Reference \*\*\*\*/

System.out.println("\n---------------------Using Method Reference---------------------");

weeks.stream().map(String::toUpperCase).forEach(System.out::println);

}

}

IV. Reference to a Constructor

Syntax :

ClassName::new

e.g.

import java.util.function.BiConsumer;

class MathOperations {

public MathOperations(int a, int b) {

System.out.println("Sum of " + a + " and " + b + " is " + (a + b));

}

}

/\*\*\*\*\* Reference To A Constructor \*\*\*\*\*/

public class MethodReferenceEx4 {

public static void main(String[] args) {

/\*\*\*\* Using Lambda Expression \*\*\*\*/

System.out.println("--------------------Using Lambda Expression----------------------");

BiConsumer<Integer, Integer> addtion1 = (a, b) -> new MathOperations(a, b);

addtion1.accept(10, 20);

/\*\*\*\* Using Method Reference \*\*\*\*/

System.out.println("\n---------------------Using Method Reference---------------------");

BiConsumer<Integer, Integer> addtion2 = MathOperations::new;

addtion2.accept(50, 20);

}

}

---------------------------------------------------------------------------------------------------------------------------

29> What is Optional? Why and how can you use it ? Explaination in detailed.

Optional is a container object that may have a null or non-null value.

Java 8 has introduced a new class Optional in java.util package.

Optional 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.

There are following advantages of Optional as -

Advantages of Java 8 Optional:

I. Null checks are not required.

II. No more NullPointerException at run-time.

III. We can develop clean and neat APIs.

IV.No more Boiler plate code

i) Use Optional.empty() to create empty optional.

Optional<Integer> possible = Optional.empty();

ii) Use Optional.of() to create optional with default non-null value. If you pass null in of(), then a NullPointerException is thrown immediately.

Optional<Integer> possible = Optional.of(5);

iii) Use Optional.ofNullable() to create an Optional object that may hold a null value. If parameter is null, the resulting Optional object would be empty (remember that value is absent; don’t read it null).

Optional<Integer> possible = Optional.ofNullable(null);

//or

Optional<Integer> possible = Optional.ofNullable(5);

Basic Example for Optional Methods Demostrations -

import java.util.Optional;

public class OptionalBasicExample {

public static void main(String[] args) {

Optional<String> gender = Optional.of("MALE");

String answer1 = "Yes";

String answer2 = null;

System.out.println("Non-Empty Optional:" + gender);

System.out.println("Non-Empty Optional: Gender value : " + gender.get());

System.out.println("Empty Optional: " + Optional.empty());

System.out.println("ofNullable on Non-Empty Optional: " + Optional.ofNullable(answer1));

System.out.println("ofNullable on Empty Optional: " + Optional.ofNullable(answer2));

// java.lang.NullPointerException

System.out.println("ofNullable on Non-Empty Optional: " + Optional.of(answer2));

}

}

Output --

Non-Empty Optional:Optional[MALE]

Non-Empty Optional: Gender value : MALE

Empty Optional: Optional.empty

ofNullable on Non-Empty Optional: Optional[Yes]

ofNullable on Empty Optional: Optional.empty

Exception in thread "main" java.lang.NullPointerException

at java.util.Objects.requireNonNull(Objects.java:203)

at java.util.Optional.<init>(Optional.java:96)

at java.util.Optional.of(Optional.java:108)

//...

Methods of Optional class :

I. isPresent() returns true if the given Optional object is non-empty. Otherwise it returns false.

public static void main(String[] args) {

Optional<String> opt = Optional.of("Ramesh");

System.out.println(opt.isPresent());

}

Output : true

II. ifPresent() performs given action if the given Optional object is non-empty. Otherwise it returns false.

public static void main(String[] args) {

Optional<String> opt = Optional.of("Ramesh");

opt.ifPresent(str -> System.out.println(str.length()));

}

Output : 6

III. orElse() return the value if present otherwise returns other.

public static void main(String[] args) {

String nullName = null;

String name = Optional.ofNullable(nullName).orElse("Ramesh");

System.out.println(name);

}

Output : Ramesh

public static void main(String[] args) {

String nullName = "Kiran";

String name = Optional.ofNullable(nullName).orElse("Ramesh");

System.out.println(name);

}

Output : Kiran

IV. orElseGet() Return the value if present, otherwise invoke other and return the result of that invocation.

public static void main(String[] args) {

String nullName = null;

String name = Optional.ofNullable(nullName).orElseGet(() -> "Ramesh");

System.out.println(name);

}

Output : Ramesh

V. orElseThrow() Return the contained value, if present, otherwise throw an exception to be created by the provided supplier.

public static void main(String[] args) {

String nullName = null;

String name = Optional.ofNullable(nullName)

.orElseThrow(IllegalArgumentException::new);

System.out.println(name);

}

Output : Exception in thread "main" java.lang.IllegalArgumentException

at java.util.Optional.orElseThrow(Unknown Source)

at OptionalDemo.main(OptionalDemo.java:10)

VI. get() If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.

public static void main(String[] args) {

Optional<String> opt = Optional.ofNullable("kiran");

String name = opt.get();

System.out.println(name);

}

Output : kiran