## **Why default method?**

For example, if several classes such as A, B, C and D implements an interface XYZ Interface then if we add a new method to the XYZ Interface, we have to change the code in all the classes(A, B, C and D) that implements this interface. In this example we have only four classes that implements the interface which we want to change but imagine if there are hundreds of classes implementing an interface then it would be almost impossible to change the code in all those classes. This is why in java 8, we have a new concept “default methods”. These methods can be added to any existing interface and we do not need to implement these methods in the implementation classes mandatorily, thus we can add these default methods to existing interfaces without breaking the code.

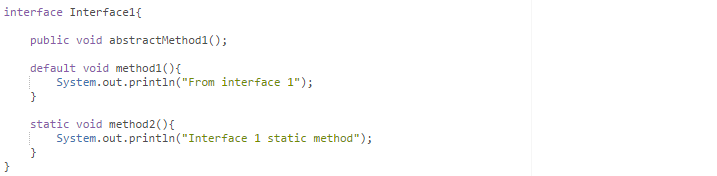
We can say that concept of default method is introduced in java 8 to add the new methods in the existing interfaces in such a way so that they are backward compatible. Backward compatibility is adding new features without breaking the old code.

**Static methods** in interfaces are similar to the default methods except that we cannot override these methods in the classes that implements these interfaces.

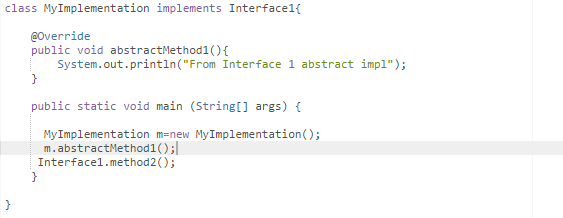
# 1. default and static methods in Interfaces:

As we know Java doesn’t support multiple inheritance for class level but which can be resolve by interface. If 2 interface contain same method signature also there may be chance for ambiguity to identify by compiler which can be resolved by default, method in interface.

**Example to Access Default and static methods.**

****

In above code interface contains all type of methods like abstract, default and static .Let’s write implementation class for interface 1.



**Here no mandatory to override default method and static method.**

Normally static method is used in interface to avoid utility class and some common logic like collection sorting etc.…2) Default method are used to write your custom implementation logic for specific class.

**Important points about java interface default methods:**

1. Java interface default methods will help us in extending interfaces without having the fear of breaking implementation classes.
2. Java interface default methods has bridge down the differences between interfaces and abstract classes.
3. Java 8 interface default methods will help us in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.
4. Java interface default methods will help us in removing base implementation classes, we can provide default implementation and the implementation classes can choose which one to override.
5. One of the major reason for introducing default methods in interfaces is to enhance the Collections API in Java 8 to support lambda expressions.
6. If any class in the hierarchy has a method with same signature, then default methods become irrelevant. A default method cannot override a method from java.lang.Object. The reasoning is very simple, it’s because Object is the base class for all the java classes. So even if we have Object class methods defined as default methods in interfaces, it will be useless because Object class method will always be used. That’s why to avoid confusion, we can’t have default methods that are overriding Object class methods.
7. Java interface default methods are also referred to as Defender Methods or Virtual extension methods.

**Important points about java interface static method:**

1. Java interface static method is part of interface; we can’t use it for implementation class objects.
2. Java interface static methods are good for providing utility methods, for example null check, collection sorting etc.
3. Java interface static method helps us in providing security by not allowing implementation classes to override them.
4. We can’t define interface static method for Object class methods, we will get compiler error as “This static method cannot hide the instance method from Object”. This is because it’s not allowed in java, since Object is the base class for all the classes and we can’t have one class level static method and another instance method with same signature.

We can use java interface static methods to remove utility classes such as Collections and move all of its static methods to the corresponding interface that would be easy to find and use

What is Lambda Expression?

Lambda expression is a new feature which is introduced in Java 8. A lambda expression is an anonymous function. A function that doesn’t have a name and doesn’t belong to any class.

It is an anonymous(Nameless) function.

Nameless

Without return type

without modifiers

Ex1:

====

public void m1(){

System.out.println("Welcome");

}

Converted into lambda Expression

()->{System.out.println("Hello");}

or

()-> System.out.println("Hello");

Note:

=====

Lambda expression contains one line curly braces are optional

eg2:

====

public void m1(int a,int b){

System.out.println(a+b);

}

Convert into lambda Expression

(a,b) -> System.out.println(a+b);

eg3:

====

public int squareIt(int n){

return n\*n;

}

convert into Lambda Expression.

(int n) ->{return n\*n;}; --1

(n) -> n\*n;

n-> n\*n; --small Expression --2

Note:

=====

1) IF you want to return the value using

return keyword the braces are mandatory

2) if you using one parameter then parenthesis is optional.

eg4:

====

public int m1(String s){

return s.length();

}

(s)->{return s.length();} ;

Or

s->s.lenght();

convert into lambda expression

s->s.length();

## **Where to use the Lambdas in Java?**

To use lambda expression, you need to either create your own functional interface or use the predefined functional interface provided by Java.

An interface with **only single abstract method** is called functional interface(or Single Abstract method interface), for example: Runnable, Callable, ActionListener Comparator,etc.

FI --Functional Interface to call lambda Expression

To use [lambda expression in Java](https://beginnersbook.com/2017/10/java-lambda-expressions-tutorial-with-examples/), you need to either create your own functional interface or use the predefined functional interface provided by Java.

While creating your own functional interface, mark it with **@FunctionalInterface** annotation, this annotation is introduced in Java 8. Although it’s optional, you should use it so that you get a compilation error if the interface you marked with this annotation is not following the rules of functional interfaces.

## **What are the rules of defining a functional interface?**

The functional interface should have **only one** abstract method. Along with the one abstract method, they can have any number of default and static methods.

Runnable ==> run()

Comparable ==> compareTo()

Comparator ==>compare()

Callable==>call()

note:

=====

An interface can contain Single Abstract Method

is called Functional Interface.

Once we write Lambda Expression to invoke

we need Functional Interfaces

can i take default methods &static methods

in Functional Interface?

yes but Exact only abstract method and

any no of default and static member

# Java Method References

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference

## **Types of Method References**

There are following types of method references in java:

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

**1. Method reference to an instance method of an object**

@FunctionalInterface

interface MyInterface{

void display();

}

public class Example {

public void myMethod(){

System.out.println("Instance Method");

}

public static void main(String[] args) {

Example obj = new Example();

// Method reference using the object of the class

MyInterface ref = obj::myMethod;

// Calling the method of functional interface

ref.display();

}

}

**2. Method reference to an static method**

@FunctionalInterface

interface MyInterface{

void display();

}

public class Example {

public static void myMethod(){

System.out.println("Instance Method");

}

public static void main(String[] args) {

// Method reference using the classname

MyInterface ref = Example::myMethod;

// Calling the method of functional interface

ref.display();

}

}

## **3. Method reference to a constructor**

@FunctionalInterface

interface MyInterface{

Hello display(String say);

}

class Hello{

public Hello(String say){

System.out.print(say);

}

}

public class Example {

public static void main(String[] args) {

//Method reference to a constructor

MyInterface ref = Hello::new;

ref.display("Hello World!");

}

}

Java8Stream():

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

Java provides a new additional package in Java 8 called java.util.stream.

This package consists of classes, interfaces, and an enum to allows

Functional-style operations on the elements.

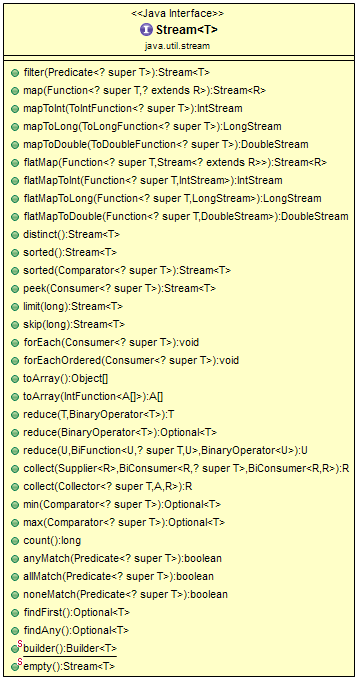
You can use stream by importing java.util.stream package in your programs.

The Stream provides the following features:

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

* Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
* Stream is functional in nature. Operations performed on a stream do not modify its source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
* Stream is lazy and evaluates code only when required.
* The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

You can use Stream to filter, collect, print, and convert from one data structure to other etc.



Stream API's Examples

Let's explore Stream APIs/methods with examples.

**1. Creating Empty Stream**

The empty() method should be used in case of the creation of an empty stream:

Stream<String> stream = Stream.empty();

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

It's often the case that the empty() method is used upon creation to avoid returning null for streams with no element:

public Stream<String> streamOf(List<String> list) {

return list == null || list.isEmpty() ? Stream.empty() : list.stream();

}

**2. Creating Stream from From Collections**

A stream can be created of any type of Collection (Collection, List, Set ):

import java.io.IOException;

import java.util.Arrays;

import java.util.Collection;

import java.util.HashSet;

import java.util.List;

import java.util.Set;

import java.util.stream.Stream;

public class StreamCreationExamples1 {

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

Collection<String> collection = Arrays.asList("JAVA", "J2EE", "Spring","Hibernate");

Stream<String> stream2 = collection.stream();

stream2.forEach(System.out::println);

List<String> list = Arrays.asList("JAVA", "J2EE", "Spring", "Hibernate");

Stream<String> stream3 = list.stream();

stream3.forEach(System.out::println);

Set<String> set = new HashSet<>(list);

Stream<String> stream4 = set.stream();

stream4.forEach(System.out::println);

}

}

Output:

JAVA

J2EE

Spring

Hibernate

JAVA

J2EE

Spring

Hibernate

JAVA

Hibernate

J2EE

Spring

**3. Creating Stream object from Arrays**

Array can be a source of a Stream or Array can be created from the existing array or of a part of an array:

import java.io.IOException;

import java.util.Arrays;

import java.util.stream.Stream;

public class StreamCreationExamples2 {

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

// Array can also be a source of a Stream

Stream<String> streamOfArray = Stream.of("a", "b", "c");

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

// creating from existing array or of a part of an array:

String[] arr = new String[] { "a", "b", "c" };

Stream<String> streamOfArrayFull = Arrays.stream(arr);

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

Stream<String> streamOfArrayPart = Arrays.stream(arr, 1, 3);

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

}

}

Output:

a

b

c

a

b

c

b

c

**4. Filtering Collection without using Stream**

In the following example, we are filtering data without using a stream. This approach we are used before the stream package was released.

First create a **Product** class, which is used in the below examples:

public class Product {

private int id;

private String name;

private float price;

public Product(int id, String name, float price) {

this.id = id;

this.name = name;

this.price = price;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public float getPrice() {

return price;

}

public void setPrice(float price) {

this.price = price;

}}

Let's first discuss without using Stream API's examples then we will create the same examples using Stream APIs

import java.util.ArrayList;

import java.util.List;

public class JavaWithoutStreamExample {

private static List < Product > productsList = new ArrayList < Product > ();

public static void main(String[] args) {

// Adding Products

productsList.add(new Product(1, "HP Laptop", 25000f));

productsList.add(new Product(2, "Dell Laptop", 30000f));

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

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

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

// Without Java 8 Stream API'S

withoutStreamAPI();

}

private static void withoutStreamAPI() {

// without Stream API's

List < Float > productPriceList = new ArrayList < Float > ();

// filtering data of list where price greater than 2500 store into productPriceList

for (Product product: productsList) {

if (product.getPrice() > 25000) {

// adding price to a productPriceList

productPriceList.add(product.getPrice());

}

}

// displaying data

for (Float price: productPriceList) {

System.out.println(price);

}

}

}

Output:

30000.0

28000.0

28000.0

90000.0

**5. Filtering Collection by using Stream**

Here, we are filtering data by using a stream. You can see that code is optimized and maintained. The stream provides fast execution.

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;

public class JavaStreamExample {

private static List < Product > productsList = new ArrayList < Product > ();

public static void main(String[] args) {

// Adding Products

productsList.add(new Product(1, "HP Laptop", 25000f));

productsList.add(new Product(2, "Dell Laptop", 30000f));

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

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

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

// With Java 8 Stream API'S

withStreamAPI();

}

private static void withStreamAPI() {

// filtering data of list

List < Float > productPriceList = productsList.stream().filter((product) -> product.getPrice() > 25000)

.map((product) -> product.getPrice()).collect(Collectors.toList());

// displaying data

productPriceList.forEach((price) -> System.out.println(price));

}

}

Output:

30000.0

28000.0

28000.0

90000.0

**6. Filtering and Iterating Collection**

In the following example, we are using the filter() method for filtering and forEach() for iterating a stream:

public class FilteringAndIteratingCollection {

public static void main(String[] args) {

List < Product > productsList = new ArrayList < Product > ();

// Adding Products

productsList.add(new Product(1, "HP Laptop", 25000 f));

productsList.add(new Product(2, "Dell Laptop", 30000 f));

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

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

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

// This is more compact approach for filtering data

productsList.stream().filter(product -> product.getPrice() == 30000)

.forEach(product -> System.out.println(product.getPrice()));

}

}

**7. Sum by using Collectors Methods**

We can also use collectors to compute a sum of numeric values.

In the following example, we are using the Collectors class and its specified methods to compute the sum of all the product prices.

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;

public class SumByUsingCollectorsMethods {

public static void main(String[] args) {

List < Product > productsList = new ArrayList < Product > ();

//Adding Products

productsList.add(new Product(1, "HP Laptop", 25000f));

productsList.add(new Product(2, "Dell Laptop", 30000f));

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

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

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

// Using Collectors's method to sum the prices.

double totalPrice3 = productsList.stream()

.collect(Collectors.summingDouble(product -> product.getPrice()));

System.out.println(totalPrice3);

}}

**8. Java Stream Example: Find Max and Min Product Price**

The following example finds min and max product prices by using stream. It provides a convenient way to find values without using the imperative approach.

import java.util.ArrayList;

import java.util.List;

public class FindMaxAndMinMethods {

public static void main(String[] args) {

List < Product > productsList = new ArrayList < Product > ();

// Adding Products

productsList.add(new Product(1, "HP Laptop", 25000f));

productsList.add(new Product(2, "Dell Laptop", 30000f));

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

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

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

// max() method to get max Product price

Product productA = productsList.stream()

.max((product1, product2) -> product1.getPrice() > product2.getPrice() ? 1 : -1).get();

System.out.println(productA.getPrice());

// min() method to get min Product price

Product productB = productsList.stream()

.max((product1, product2) -> product1.getPrice() < product2.getPrice() ? 1 : -1).get();

System.out.println(productB.getPrice());

}

}

Output:

90000.0

25000.0

**9. Java Stream Example: Convert List into Set**

import java.util.ArrayList;

import java.util.List;

import java.util.Set;

import java.util.stream.Collectors;

public class ConvertListToSet {

public static void main(String[] args) {

List < Product > productsList = new ArrayList < Product > ();

// Adding Products

productsList.add(new Product(1, "HP Laptop", 25000 f));

productsList.add(new Product(2, "Dell Laptop", 30000 f));

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

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

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

// Converting product List into Set

Set < Float > productPriceList = productsList.stream().filter(product -> product.getPrice() < 30000)

.map(product -> product.getPrice()).collect(Collectors.toSet());

System.out.println(productPriceList);

}

}

**10. Java Stream Example: Convert List into Map**

public class ConvertListToMap {

public static void main(String[] args) {

List < Product > productsList = new ArrayList < Product > ();

// Adding Products

productsList.add(new Product(1, "HP Laptop", 25000 f));

productsList.add(new Product(2, "Dell Laptop", 30000 f));

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

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

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

// Converting Product List into a Map

Map < Integer, String > productPriceMap = productsList.stream()

.collect(Collectors.toMap(p -> p.getId(), p -> p.getName()));

System.out.println(productPriceMap);

}

}

**11. Using Method References in Stream Examples**

public class MethodReferenceInStream {

public static void main(String[] args) {

List < Product > productsList = new ArrayList < Product > ();

// Adding Products

productsList.add(new Product(1, "HP Laptop", 25000 f));

productsList.add(new Product(2, "Dell Laptop", 30000 f));

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

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

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

List < Float > productPriceList = productsList.stream()

.filter(p -> p.getPrice() > 30000) // filtering data

.map(Product::getPrice) // fetching price by referring getPrice method

.collect(Collectors.toList()); // collecting as list

System.out.println(productPriceList);

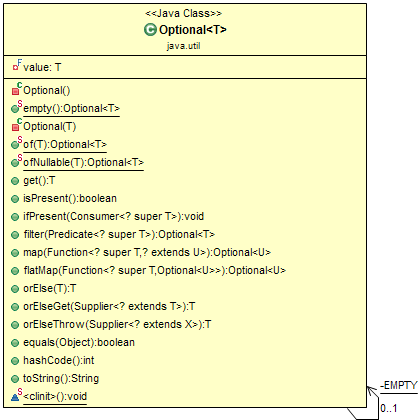
}

}

**Java 8 Optional Class**

Java introduced a new class Optional in JDK 8. It is a public final class and is used to deal with NullPointerException in Java applications. You must import java.util package to use this class. It provides methods that are used to check the presence of a value for the particular variable.

The purpose of the class is to provide a type-level solution for representing optional values instead of using null references.



**Advantages of Java 8 Optional**

* Null checks are not required.
* No more NullPointerException at run-time.
* We can develop clean and neat APIs.
* No more Boilerplate code

**Optional Class Simple Example**

The Optional.ofNullable() method returns a Non-empty Optional if a value present in the given object. Otherwise returns empty Optional.

Optional.empty() method is useful to create an empty Optional object.

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));

}

}

**Usage of Optional class APIs/Methods**

**isPresent()** Optional class API

The isPresent() method returns an Optional with the specified present non-null value.

Example :

private static void isPresentOptionalAPI() {

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

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

}

**empty() Optional class API**

An empty() static method returns an empty Optional instance. No value is present for this Optional.

Example :

// Returns an Optional with the specified present non-null value.

private static void createEmptyOptionalObject() {

Optional<String> empty = Optional.empty();

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

// Optional object with the static of API:

String name = "Ramesh";

Optional.of(name);

}

**ifPresent() Optional class API**

If a value is present, invoke the specified consumer with the value, otherwise, do nothing.

Example :

private static void ifPresentOptionalAPI() {

// The ifPresent API enables us to run some code on the wrapped value if it is

// found to be non-null.

// Before Optional, we would do something like this:

String name = "Ramesh";

if (name != null) {

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

}

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

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

}

**orElse() Optional class API**

Return the value if present otherwise returns another.

Example :

private static void orElseOptionalAPI() {

// With orElse, the wrapped value is returned if it is present and the argument

// given to

// orElse is returned if the wrapped value is absent

String nullName = null;

// If a value is present, invoke the specified consumer with the value, otherwise

// do nothing.

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

System.out.println(name);

}

**orElseGet() Optional class API**

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

Example :

private static void orElseGetOptionalAPI() {

String nullName = null;

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

System.out.println(name);

}

**orElseThrow() Optional class API**

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

Example :

private static void orElseThrowOptionalAPI() {

String nullName = null;

String name = Optional.ofNullable(nullName)

.orElseThrow(IllegalArgumentException::new);

System.out.println(name);

}

**get() Optional class API**

If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.

Example :

private static void getOptionalAPI() {

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

String name = opt.get();

System.out.println(name);

}

**Complete Example for Reference**

import java.util.Optional;

public class OptionalClassExamples {

public static void main(String[] args) {

isPresentOptionalAPI();

createEmptyOptionalObject();

createEmptyOptionalObjectWithStaticAPI();

ifPresentOptionalAPI();

orElseOptionalAPI();

orElseOptionalAPI();

orElseGetOptionalAPI();

orElseThrowOptionalAPI();

getOptionalAPI();

}

// Returns an Optional with the specified present non-null value.

private static void isPresentOptionalAPI() {

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

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

}

// Returns an Optional with the specified present non-null value.

private static void createEmptyOptionalObject() {

Optional < String > empty = Optional.empty();

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

// Optional object with the static of API:

String name = "Ramesh";

Optional.of(name);

}

private static void createEmptyOptionalObjectWithStaticAPI() {

String name = "baeldung";

Optional.of(name);

}

// If a value is present, invoke the specified consumer with the value, otherwise do

// nothing.

private static void ifPresentOptionalAPI() {

// The ifPresent API enables us to run some code on the wrapped value if it is

// found to be non-null.

// Before Optional, we would do something like this:

String name = "Ramesh";

if (name != null) {

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

}

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

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

}

// If a value is present, invoke the specified consumer with the value, otherwise do

// nothing.

private static void orElseOptionalAPI() {

// With orElse, the wrapped value is returned if it is present and the argument

// given to

// orElse is returned if the wrapped value is absent

String nullName = null;

// If a value is present, invoke the specified consumer with the value, otherwise

// do nothing.

//

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

System.out.println(name);

}

private static void orElseGetOptionalAPI() {

String nullName = null;

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

System.out.println(name);

}

private static void orElseThrowOptionalAPI() {

// This will throw exception

String nullName = null;

String name = Optional.ofNullable(nullName)

.orElseThrow(IllegalArgumentException::new);

System.out.println(name);

}

private static void getOptionalAPI() {

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

String name = opt.get();

System.out.println(name);

}

}