**Does Interface and functional interface are same** ?

No, interfaces and functional interfaces are not the same, although functional interfaces are a subset of interfaces in Java.

Here's the distinction: **Functional interfaces** have a specific constraint of containing **only one abstract method**, while **regular interfaces in Java can contain any number of abstract methods,** default methods, and static methods (with Impl).

We cannot Create a Object for Java7 Interface, as it doesn’t contain any Impl, where as for Functional Interface can be Instantiated using Lambda expressions or method reference

**Functional Interface :**

An Interface that contains exactly **one abstract method** is known as functional interface. It can **have any number of default, static methods(with impl)** but can contain only one abstract method.

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.

**@FunctionalInterface** annotation is added so that we can mark an interface as functional interface.

It is not mandatory to use it, but it’s best practice to use it with functional interfaces to avoid addition of extra abstract methods accidentally. If the interface is annotated with @FunctionalInterface annotation and we try to have more than one abstract method, it throws compiler error.

Functional Interface are mainly introduced to solve the problem (Backward Compatibility meaning a new feature or version should support older version without breaking) like, if we want to add a new feature or method in an interface then we should add that as abstract method in interface then all the child classes need to break the existing contract and need to provide implementation, instead of doing that if we are able to add implementation directly in interface then we don’t need to break or modify child class code. during java 8 development when they tried to add new feature in collection framework, they had this issue so they introduced this feature.

Functional Interface facilitate the use of lambda expressions and method references, which in turn enhance code readability, support functional programming paradigms, and enable features like the Streams API for efficient collection processing. They represent a fundamental aspect of modern Java programming, promoting concise, readable, and maintainable code.

**Some Built-in Java Functional Interfaces**

Since Java SE 1.8 onwards, there are many interfaces that are converted into functional interface. All these interfaces are annotated with @FunctionalInterface. These interfaces are as follows –

* **Runnable –>** This interface only contains the run() method.
* **Comparable –>** This interface only contains the compareTo() method.
* **ActionListener –>** This interface only contains the actionPerformed() method.
* **Callable –>** This interface only contains the call() method.

**Java SE 8 included four main kinds of functional interfaces in java.util.function package**which can be applied in multiple situations. These are:

1. Consumer
2. Predicate
3. Function
4. Supplier

Amidst the previous four interfaces, the first three interfaces,i.e., Consumer, Predicate, and Function, likewise have additions that are provided beneath –

1. Consumer -> Bi-Consumer
2. Predicate -> Bi-Predicate
3. Function -> Bi-Function, Unary Operator, Binary Operator

**Lambda expressions**

Lambda expressions enable Java’s object-oriented programming world with functional programming. It **provides an implementation of an interface which has a functional interface**. When lambda functions are used, we don’t need to define the method by providing the implementation. In fact, lambda expressions are treated as a function.

The basic syntax of Lambda expression is **(argument) → (body)**

In General Object oriented program every thing is represented as object only we cannot do anything with out object, it means if you want to print a simple hello word also we need to create a class , object and then hello world method. To avoid this in java 8 they introduced functional programming we can achieve this with functional interface and lambda expression

Example : if we want to print a helloword without functional interface we need to create a Greet interface with printGreet() abstract method then we need to provide implementation for printGreet() method in our own class like GreetImpl then create a object with that reference we call printGreet() method . But if we use functional interface we can just provide implementation for that without creating object or class

Greet g=() -> System.out.println(“hello word greet message”) g.printgreet(); Here we are providing implementation with lambda expression ()-> lambda expression or lambda function is a anonymous with without any method name, lambda expression must be a type of functional interface, it means as we assign some value to variable name of type int or String and we can pass int or string values as argument we can do same thing with lambda expression we can assign a function (lambda expression) to a variable of type FunctionalInterface and we can pass functional interface as argument but receiver method must have parameter type as functional interface.

**Functional programming reference** : [**https://www.geeksforgeeks.org/functional-programming-in-java-with-examples/**](https://www.geeksforgeeks.org/functional-programming-in-java-with-examples/)

Public **interface** Funtest{

**void** m2(String s,**int** x);

}

**Example** 1: varible f holding the function below Funtest f=(s,a)-> System.***out***.println(s+" "+a );

f.m2("hello world",10);

**Example** 2: passing function as argument

Public **class** Test{

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

*printmethod*((s,a)-> System.***out***.println(s+" "+a ));

}

**static** **void** printmethod(Funtest fun){

fun.m2("hello world",10);

}

}

Example 3:

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

Funtest f=(s)-> s.length();

System.***out***.println(f.m2("hello world"));

}

**public** **interface** Funtest {

**int** m2(String s);

}

Basic syntax is ()-> logic // for single line code For multiple line code : () -> { logic/////// } above two are without parameters we can have parameters and return type also for void method we don’t need to mention any return type example (a,b)-> a+b; // here a+b is returns value of type int in functional interface abstract method must be like **public int cal (int a,int b); another example for lambda expression**

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

Funtest f= (a,b)-> {

System.***out***.println(a+b);

System.***out***.println(a\*b);

**int** z=a+b;

**return** z;

};

System.***out***.println(f.cal(10, 20));

}

When we use single parameters for lambda expression we don’t even required to specify () bracket symbol also

**Example Funtest f=s -> s.length();**

**System.*out*.println(f.t("hello world"));**

int t(String s);// we should add this method in interface

point to remember in lambda expression left side will be parameter of input and right side to -> arrow symbol will be return type value

most important thing is a lambda expression must be assigned to a functional interface type variable or passed to argument to type of functional interface.

**Functional interfaces**

1. Consumer : Accepts **a single argument** and returns **no result --void** accept(T t); andThen
2. Predicate : which **accepts an argument** and **returns a boolean**. Usually, it used to apply in a filter for a collection of objects.-- **boolean** test(T t); and or negate
3. Function : Accepts **one argument** and **returns a result. --** R apply(T t);
4. Supplier :  it takes **no arguments** and returns **a result. --**T get();

**Convert String to Integer (below code same with Lamda and method ref)**

Function<String, Integer> stoi=(String s)->Integer.*valueOf*(s); //with Lambda

Function<String, Integer> stoix=Integer::*valueOf*; //with Method ref

stoix.apply("20");

System.***out***.println(stoi.apply("10"));

# **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:

We can call a method by using :: symbol if we want to call static method we need to use class name , if we want to call instance method use object reference and if we want to call constructor use class name and new keyword, Note: for these method reference concept we must use functional interface , method reference will be reference of functional interface , method reference is like lamda expression.

1. **Reference to a static method**: This type of method reference refers to a static method. It is denoted by the syntax **ContainingClass::staticMethodName**.
2. **Reference to an instance method of a particular object**: This type of method reference refers to an instance method of a particular object. It is denoted by the syntax **objectReference::instanceMethodName**.
3. **Reference to an instance method of an arbitrary object(Specific Object) of a particular type**: This type of method reference refers to an instance method of an arbitrary object of a particular type. It is denoted by the syntax **ContainingType::instanceMethodName**.
4. **Reference to a constructor**: This type of method reference refers to a constructor. It is denoted by the syntax **ClassName::new**.

**Note: Main difference between 2nd and 3rd point is 2nd point refers to particular object like Ex: Person P= new Persion(“Teja”) // P::getName (only one obj type)**

**Where as 3rd point refers to Arrays.asList(new Persion(“Teja”), new Persion(“raja”)).stream().map(Person::getName) here person Type but each new object as different.**

**Syntax**: Classname::staticMethodName

Consumer<String> z= System.***out***::println;

z.accept("hello world");

**Syntax**:  ObjectReference::saySomething;

Main(){

Consumer<String> ss=tt::dummy;

ss.accept("hello dummy");

}

**public** **void** dummy(String x) {

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

}

**Syntax**: ClassName::**new**  .

Tester(String x){

System.***out***.println(x +"inside constr");

}

Consumer<String> st=Tester::**new**;

st.accept("hello dummy call const");

}

**Syntax for lambda expression**

Consumer<String> z= x-> System.***out***.println(x)

z.accept("hello world");

**Below is Example of Reference to an instance method of an arbitrary object of a particular type**

List<Person> people = Arrays.asList(**new Person**("Alice"),**new Person**("Bob"),new Person("Charlie")); // Method reference to an instance method of an arbitrary object of a particular type

people.stream().map(**Person**::getName).forEach(System.out::println);

// Here, Person::getName is a method reference

# **Java Optional Class**

The **Optional** class in Java 8 is a **container object** that may or may not contain a non-null value. It was introduced to address the problem of null references and to provide a more explicit and safer way to represent the absence of a value.

Here are some key points about the **Optional** class:

1. **Encourages explicit handling of null**: **Optional** encourages developers to handle null values explicitly by providing methods for checking the presence of a value (**isPresent()**), accessing the value safely (**get()**), or providing a default value if the value is absent (**orElse()**).
2. **Avoids NullPointerExceptions (NPEs)**: By using **Optional** instead of returning null from methods or dereferencing variables, developers can reduce the likelihood of encountering NPEs, as the need to explicitly check for null is enforced by the API.
3. **Promotes clearer and safer code**: Code using **Optional** tends to be clearer and safer because it explicitly communicates whether a value is present or absent, reducing the risk of accidentally dereferencing null values without checking for null.
4. **Not a replacement for all null occurrences**: It's important to use **Optional** judiciously and not as a blanket replacement for all occurrences of null. It's generally recommended to use **Optional** for return values of methods or parameters that may be absent, rather than as fields in classes or elements in collections.

Overall, the **Optional** class in Java 8 provides a more structured and safer way to handle the absence of values, leading to more robust and null-safe code.

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. IsPresent , get , filter are mostly used methods**

**In Optional class we mainly use** Optional.of() and Optional.ofNullable() methods we use Optional.of() when we are sure its not null like Optional.of(getResult) we use if present method to check if its present we handle success or else failure case in case if it is null Optional.of(null) it will throw null pointer exception, we use Optional.ofNullable(getResult) when we are not sure about value is null or not if incase value is null Optional.ofNullable(null) it will give us empty optional object, if don’t want to throw any error just return empty result we use this

Optional<String> s=Optional.*of*("").filter(x->x.startsWith("a"));

**if**(s.isPresent()) {

System.***out***.println("ok");

}

**else** {

System.***out***.println("not ok");

}

1. **of(T value)**:
   * Creates an **Optional** instance containing the specified non-null value.
   * Throws a **NullPointerException** if the provided value is null.
2. **ofNullable(T value)**:
   * Creates an **Optional** instance containing the specified value if it is non-null.
   * If the provided value is null, creates an empty **Optional** instance.
3. **empty()**:
   * Returns an empty **Optional** instance.
4. **get()**:
   * Returns the value if present, otherwise throws a **NoSuchElementException**.
   * It's recommended to avoid using **get()** and prefer other methods like **orElse()** or **orElseThrow()** to handle absent values safely.
5. **isPresent()**:
   * Returns true if a value is present, otherwise false.
6. **ifPresent(Consumer<? super T> consumer)**:
   * Performs the given action if a value is present.
7. **orElse(T other)**:
   * Returns the value if present, otherwise returns the specified default value.
8. **orElseGet(Supplier<? extends T> other)**:
   * Returns the value if present, otherwise calls the specified supplier to generate a default value.
9. **orElseThrow(Supplier<? extends X> exceptionSupplier)**:
   * Returns the value if present, otherwise throws an exception created by the provided supplier.
10. **map(Function<? super T,? extends U> mapper)**:
    * If a value is present, applies the given mapping function to it and returns the result wrapped in an **Optional**, otherwise returns an empty **Optional**.
11. **flatMap(Function<? super T,Optional<U>> mapper)**:
    * If a value is present, applies the given mapping function to it and returns the result, otherwise returns an empty **Optional**.

[**https://www.javatpoint.com/java-8-optional**](https://www.javatpoint.com/java-8-optional)

# **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.

1. **default** **void** forEach(Consumer<**super** T>action)
2. List<String> gamesList = **new** ArrayList<String>();
3. gamesList.add("Football");
4. gamesList.add("Cricket");
5. gamesList.add("Chess");
6. gamesList.add("Hocky");
7. System.out.println("------------Iterating by passing method reference---------------");
8. gamesList.forEach(System.out::println);

// gamesList.forEach(games -> System.out.println(games));

**forEachOrdered().** It is used to iterate elements in the order specified by the stream.

# **Default Methods** In Java 8

Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

The default methods were introduced to provide backward compatibility so that existing interfaces can use the lambda expressions without implementing the methods in the implementation class. Default methods are also known as **defender methods**or **virtual extension methods**.

**Important Points:**

1. Interfaces can have default methods with implementation in Java 8 on later.
2. Interfaces can have static methods as well, similar to static methods in classes.
3. Default methods were introduced to provide backward compatibility for old interfaces so that they can have new methods without affecting existing code.

**For example**, ‘List’ or ‘Collection’ interfaces do not have ‘forEach’ method declaration. Thus, adding such method will simply break the collection framework implementations. Java 8 introduces default method so that List/Collection interface can have a default implementation of forEach method, and the class implementing these interfaces need not implement the same.

## **Multiple Defaults**

With default functions in interfaces, there is a possibility that a class is implementing two interfaces with same default methods. The following code explains how this ambiguity can be resolved.

First solution is to create an own method that overrides the default implementation.

Second solution is to call the default method of the specified interface using super.

public class car implements **vehicle**, fourWheeler {

public void print() {

**vehicle.super.print();**

}

}

https://www.tutorialspoint.com/java8/java8\_default\_methods.htm

# **Stream In Java**

Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.  
The features of Java stream are –

* A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Different Operations On Streams-  
**Intermediate Operations:**

**Intermediate Operations**: Intermediate operations are operations that transform or manipulate the elements of a stream. **These operations are typically lazy, meaning they do not process the elements of the stream until a terminal operation is invoked.** Intermediate operations are chainable, meaning you can apply multiple intermediate operations sequentially on a stream. Some common intermediate

1. **map:**map will transform one input to another form example if we have Stream of strings we want to know length of each string in stream we use **Array.asList(“teja”,”naik”).stream().map(String::length). .collect(Collectors.toList())**

The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.  
List number = Arrays.asList(2,3,4,5);  
List square = number.stream().map(x->x\*x).collect(Collectors.toList());

1. **filter:** Filter will gives us required result based on condition (filter data from stream as per our requirement ) like filter Strings only starts with “S”

**Arrays.asList("sam","dam").stream().filter(x->x.startsWith("s")) .collect(Collectors.toList());**

The filter method is used to select elements as per the Predicate passed as argument.  
List names = Arrays.asList("Reflection","Collection","Stream");  
List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());

1. **sorted:** The sorted method is used to sort the stream.  
   List names = Arrays.asList("Reflection","Collection","Stream");  
   List result = names.stream().sorted().collect(Collectors.toList());
2. **flatMap**
3. **distinct**
4. **peek**
5. **limit**
6. **skip**

// Create a stream of integers **from 1 to 10**

Stream<Integer> stream = Stream.iterate(1, n -> n + 1).**limit(10);**

**// Skip the first 5 elements** of the stream and print the rest

**stream.skip(5).**forEach(System.out::println);

**Terminal Operations:**

Terminal operations are operations that produce a result or side-effect from a stream. When a terminal operation is applied to a stream, it triggers the execution of the entire stream pipeline, causing the intermediate operations to be evaluated. After the terminal operation is executed, the stream is considered consumed and cannot be reused. Some common terminal operations

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.  
   List number = Arrays.asList(2,3,4,5,3);  
   Set square = number.stream().map(x->x\*x).collect(Collectors.toSet());
2. **forEach:** The forEach method is used to iterate through every element of the stream.  
   List number = Arrays.asList(2,3,4,5);  
   number.stream().map(x->x\*x).forEach(y->System.out.println(y));
3. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.  
   The reduce method takes a BinaryOperator as a parameter.

List number = Arrays.asList(2,3,4,5);  
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

Here ans variable is assigned 0 as the initial value and i is added to it .

anyMatch, allMatch, nonmatch, findFirst, findany, count, min, max

**Important Points/Observations:**

1. A stream consists of source followed by zero or more intermediate methods combined together (pipelined) and a terminal method to process the objects obtained from the source as per the methods described.
2. Stream is used to compute elements as per the pipelined methods without altering the original value of the object.

# **Java Collectors**

Collectors is a final class that extends Object class. It provides reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

Java Collectors class provides various methods to deal with elements

https://www.geeksforgeeks.org/java-collectors/