**Functional Programming:**

Functional programming is a programming paradigm in which we try to bind everything in pure mathematical functions style. It is a declarative type of programming style. Its main focus is on “what to solve” in contrast to an imperative style where the main focus is “how to solve”. It uses expressions instead of statements. An expression is evaluated to produce a value whereas a statement is executed to assign variables. Those functions have some special features discussed below.

**Functional Programming is based on Lambda Calculus:**   
Lambda calculus is a framework developed by Alonzo Church to study computations with functions. It can be called as the smallest programming language in the world. It gives the definition of what is computable. Anything that can be computed by lambda calculus is computable.

**Concepts of functional programming:**

* [**Pure functions**](https://www.geeksforgeeks.org/pure-functions/)
* [**Recursion**](https://www.geeksforgeeks.org/recursion/)
* **Referential transparency**
* **Functions are First-Class and can be Higher-Order**
* **Variables are Immutable**

**Pure functions:**These functions have two main properties. First, they always produce the same output for same arguments irrespective of anything else.   
Secondly, they have no side-effects i.e. they do not modify any arguments or local/global variables or input/output streams.   
Later property is called immutability. The pure function’s only result is the value it returns. They are deterministic.

The output of a pure function depends only on

1. Its input parameters and
2. Its internal algorithm

Programs done using functional programming are easy to debug because pure functions have no side effects or hidden I/O. Pure functions also make it easier to write parallel/concurrent applications. When the code is written in this style, a smart compiler can do many things – it can parallelize the instructions, wait to evaluate results when needing them, and memorize the results since the results never change as long as the input doesn’t change.

**Note:** **Consumer is not a pure function**  
**example of the pure function:**

sum(x, y) // sum is function taking x and y as arguments

return x + y // sum is returning sum of x and y without changing them

**Recursion:**There are no “for” or “while” loop in functional languages. Iteration in functional languages is implemented through recursion. Recursive functions repeatedly call themselves, until it reaches the base case.   
**example of the recursive function:**

fib(n)

if (n <= 1)

return 1;

else

return fib(n - 1) + fib(n - 2);

**Referential transparency:** In functional programs variables once defined do not change their value throughout the program. Functional programs do not have assignment statements. If we have to store some value, we define new variables instead. This eliminates any chances of side effects because any variable can be replaced with its actual value at any point of execution. State of any variable is constant at any instant.

**Example:**

x = x + 1 // this changes the value assigned to the variable x.

// So the expression is not referentially transparent.

An expression is referentially transparent if it can be replaced with its corresponding value without changing the program's behavior.

If all functions involved in the expression are pure and immutable functions, then the expression is referentially transparent. As a result of referential transparency, we produce context-free code. This means the functions can be executed in any order to explore optimization possibilities.

**Example:**

public class Main {

private static int sum(int a, int b) {

System.out.printf("sumReferential: Adding %d and %d%n", a, b);

return a + b;

}

private static int sumReferential(int a, int b) {

return a + b;

}

public static void main(String[] args) {

sumReferential(1, sumReferential(2, 3));

sum(1, sum(2, 3));

}

}

**Output:**

sumReferential: Adding 2 and 3

sumReferential: Adding 1 and 5

**Explanation:**

In this example replacing sumReferential(2, 3) with the values (1,5) will give the same result 6. But if we replace sum(2,3) with (1,5), we will miss the printf statement inside sum() method. So, sum() is not referentially transparent.

So, here sumReferential() can take (2,3) or (1,5) as arguments to produce same output 6 which makes it refereniatl transparent.

**Functions are First-Class Citizens:**First-class functions are treated as first-class variable. The first class variables can be passed to functions as parameter, can be returned from functions or stored in data structures.

Example:

show\_output(f) // function show\_output is declared taking argument f

// which are another function

f(); // calling passed function

print\_gfg() // declaring another function

print("hello gfg");

show\_output(print\_gfg) // passing function in another function

**Example Java Program:**

class Main {

// calculate the sum

public int add(int a, int b) {

// calculate sum

int sum = a + b;

return sum;

}

// calculate the square

public void square(int num) {

int result = num \* num;

System.out.println(result); // prints 576

}

public static void main(String[] args) {

Main obj = new Main();

// call the square() method

// passing add() as an argument

obj.square(obj.add(15, 9));

}

}

**Functions can be Higher-Order:**

A function that accepts one or more functions as arguments and/or returns a function as its output is referred to as a higher-order function.

public class FunctionalProgrammingFeatures {  
  
 //applyOperation(..) is higher order pure function  
 public static int applyOperation(int a, int b, IntBinaryOperator operation) {  
 return operation.applyAsInt(a, b);  
 }  
 static void callHigherOrderPureFunction()  
 {  
 System.*out*.println("Higher Order Pure Function Example");  
 IntBinaryOperator add = Integer::*sum*;  
 IntBinaryOperator subtract = (a, b) -> a - b;  
 IntBinaryOperator multiply = (a, b) -> a \* b;  
  
 System.*out*.println(*applyOperation*(10,20,add));  
 System.*out*.println(*applyOperation*(10,20,subtract));  
 System.*out*.println(*applyOperation*(10,20,multiply));  
 }  
}

**Variables are Immutable:**In functional programming, we can’t modify a variable after it’s been initialized. We can create new variables – but we can’t modify existing variables, and this really helps to maintain state throughout the runtime of a program. Once we create a variable and set its value, we can have full confidence knowing that the value of that variable will never change.

**Advantages and Disadvantages of Functional programming**

**Advantages:**

1. Pure functions are easier to understand because they don’t change any states and depend only on the input given to them. Whatever output they produce is the return value they give. Their function signature gives all the information about them i.e. their return type and their arguments.
2. The ability of functional programming languages to treat functions as values and pass them to functions as parameters make the code more readable and easily understandable.
3. Testing and debugging is easier. Since pure functions take only arguments and produce output, they don’t produce any changes don’t take input or produce some hidden output. They use immutable values, so it becomes easier to check some problems in programs written uses pure functions.
4. It is used to implement concurrency/parallelism because pure functions don’t change variables or any other data outside of it.
5. It adopts lazy evaluation which avoids repeated evaluation because the value is evaluated and stored only when it is needed.

**Disadvantages:**

1. Sometimes writing pure functions can reduce the readability of code.
2. Writing programs in recursive style instead of using loops can be bit intimidating.
3. Writing pure functions are easy but combining them with the rest of the application and I/O operations is a difficult task.
4. Immutable values and recursion can lead to decrease in performance.

**Applications:**

* It is used in mathematical computations.
* It is needed where concurrency or parallelism is required.

**Functional Programming Techniques:**

* Functional Chaining
* Functional Composition
* Closures
* Currying
* Lazy Evaluation
* Tail Call Optimization

1. **Functional Chaining:**

**Consumer.java**

package functionaProgrammingTechniques;  
  
import java.util.Objects;  
  
@FunctionalInterface  
  
public interface Consumer<T> {  
 void accept(T t);  
 default Consumer<T> thenAccept(Consumer<T> next){  
 Objects.*requireNonNull*(this);  
 Objects.*requireNonNull*(next);  
 return (T t)->{  
 this.accept(t);  
 next.accept(t);  
 };  
 }  
}

**Chaining.java**

package functionaProgrammingTechniques;  
  
import java.util.function.Function;  
  
public class Chaining {  
 static void chaningDemo(){  
  
 Consumer<String> c1 = s -> System.*out*.println(s);  
 Consumer<String> c2 = System.*out*::println;  
  
 c1.accept("Hello");  
 c2.accept("Hi");  
 Consumer<String> c3 = s -> {  
 c1.accept(s);  
 c2.accept(s);  
 };  
 c3.accept("Hello");  
 Consumer<String> c4=c1.thenAccept(c2);  
 c4.accept("Praneetha");  
  
 Function<Integer,Integer> f1=s->s+2;  
 Function<Integer,Integer> f2=s->s\*2;  
  
 Function<Integer,Integer> f3=f1.andThen(f2);  
 System.*out*.println(f3.apply(10));  
 }  
}

1. **Composition:**

**Square.java**

package functionaProgrammingTechniques;  
  
public class Square {  
 private int area;  
 public int getArea(){  
 return area;  
 }  
 public void setArea(int area){  
 this.area=area;  
 }  
}

**Function.java**

package functionaProgrammingTechniques;  
  
public interface Function<T,R>{  
 R apply(T t);  
 default <V> Function<V,R> compose(Function<V, T> fun1){  
 return (V v)-> this.apply(fun1.apply(v));  
 }  
}

**FunctionComposition.java**

package functionaProgrammingTechniques;  
  
public class FunctionComposition {  
 static void functionalCompositionDemo() {  
 System.*out*.println("Functional Composition Demo");  
 Function<Square,Integer> fun1=s->s.getArea();  
 Function<Integer,Double> fun2=area->Math.*sqrt*(area);  
 fun2.compose(fun1);  
 Function<Square,Double> getSide=fun2.compose(fun1);  
 Square s=new Square();  
 s.setArea(100);  
 Double side=getSide.apply(s);  
 System.*out*.println("Side of square: "+side);  
 }  
}

1. **Closure:**

In the most traditional sense, a closure is a function object that has access to variables from its containing scope, even when the function is called outside that scope.

The two key characteristics of a closure are:

1. A function or reference type that can get stored and passed around like an instance of an object. This can be an anonymous function or lambda expression.
2. The ability to access variables from its containing lexical scope, which means the function "closes over" the scope, hence the term "closure."

"Closure" in programming languages comes from the word "enclose", where a function has an "enclosed" scope. When a function "closes over" variables, it refers to the behavior of a function or method **retaining access to variables from an outer or parent scope, even after the outer function has finished its execution**. These variables are part of the function’s "closed" or "enclosed" lexical scope.

However, do note that the Java lambda expressions are not "true" closures, since the variables from the enclosing scope that they can access must be effectively final, i.e., their value should not change after their initial assignment. This is not a requirement in some other languages that support true closures.

public class JavaClosure2 {  
 static void javaClosureDemo(){  
 int num = 10; // effectively final local variable  
 Function<Integer, Integer> multiply = (x) -> x \* num; // Lambda expression using 'num'.  
 System.*out*.println(multiply.apply(2)); // Outputs 20.  
 }  
}

In this example, multiply is a lambda function or a closure-like construct that multiplies a number x with num. num is defined outside the lambda expression’s scope but still accessible within the lambda expression body. In this manner, we could say that the lambda expression is "closing over" the variable num.

**Example2:**

public class JavaClosures {  
 static void javaClosure()  
 {  
 int val=111;  
 Task lambda=()->{  
 System.*out*.println(val);  
 };  
 }  
 private static void printValue(Task lambda){  
 lambda.doTask();  
 }  
}

Here, this ability to remember and have access to its surrounding state (the val variable in this case) even when it is outside of its lexical scope is considered a type of closure behavior.

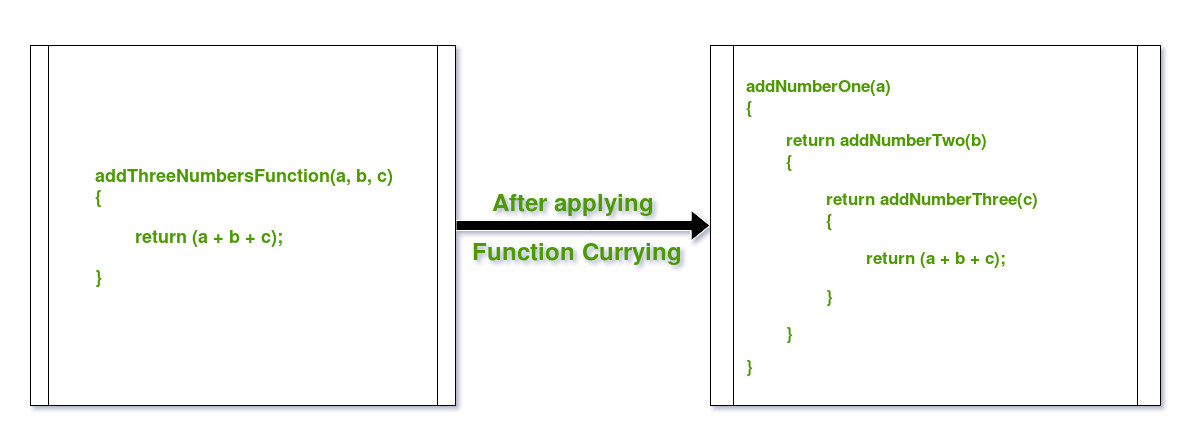
While Java doesn’t directly support closures, lambda expressions and anonymous classes introduced in the language can emulate closure-like behavior. These are able to access variables defined in the enclosing scope, yet there are certain limitations.

For both lambda expressions and anonymous classes, they can access:

* Static variables.
* Instance variables.
* Method parameters.
* Local variables, if they are declared final or effectively final, i.e., their values don't change after initialization.

1. **Currying:**

**Function Currying** is a concept of **breaking a function with many arguments into many functions with single argument in such a way, that the output is same.** In other words, its a technique of simplifying a multi-valued argument function into single-valued argument multi-functions.



Currying breaks down higher order functions into a series of smaller cascaded functions which take in one argument and return a function except for the last cascaded function which returns the desired value.

**Example:**

public class Currying {  
 static void curryingDemo(){  
 System.*out*.println("Currying Example");  
 Function<Integer,Function<Integer,Integer>> fun2=u->v->u+v;  
 Function<Integer,Function<Integer,Function<Integer,Integer>>>

fun3=u->v->w->u+v+w;  
 Integer sum=fun2.apply(1).apply(2);  
 System.*out*.println("sum: "+sum);  
 Integer sumOf3Variables=fun3.apply(1).apply(2).apply(3);  
 }  
}

|  |  |
| --- | --- |
| **Functional Programming** | **OOP** |
| Immutable data | Mutable data |
| Declarative Programming | Imperative Programming |
| Focuses on the “what” of a problem | Focuses on the “how” of a problem |
| Uses recursions, avoids loops | Uses loops |
| Supports Parallel Programming | Not suitable for Parallel Programming |
| Execution order of statements is not very important | Execution order of statements is important |

1. **Lazy Evaluation:**

Lazy evaluation in Java functional programming is a strategy where the evaluation of an expression is delayed until its value is actually required. It avoids repeated evaluation as the value is cached or 'memoized' after its initial evaluation and subsequent requests use the cached result without repeating the calculation.

1. **Tail Call Optimization:**

**Lambda:**

In Java, Lambda expressions basically express instances of functional interfaces (An interface with a single abstract method is called a functional interface). Lambda Expressions in Java are the same as lambda functions which are the short block of code that accepts input as parameters and returns a resultant value.

**Example: Creating Thread without Lambda**

public static void main(String[] args) {  
   
 Thread thread=new Thread(new Runnable() {  
 @Override  
 public void run() {  
 System.out.println("Thread in Run");  
 }  
 });  
}

A function has 4 properties:

1. Name of the function
2. Parameters List
3. Body of the function
4. Return Type

Here, We are going to remove anonymous inner class because thread class accepts Runnable and compiler knows it.

Next, we are going to remove method name run, because Runnable interface has single abstract method named run and compiler can detect that too.

We can remove access modifier also because in interface method needs to be public if someone needs to access it out.

We can remove the return type, because compiler can detect return type beyond java8 using type inference.

By putting -> between () and body we can convert the above function into lambda.

**Function Using Lambda:**

Thread thread=new Thread(  
 ()-> {  
 System.out.println("Thread started");  
 }  
);  
thread.start();

**Predefined Functional Interfaces:**

**Java SE 8 included four main kinds of functional interfaces**which can be applied in multiple situations as mentioned below:

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

|  |  |  |
| --- | --- | --- |
| **Functional interface** | **Functional descriptor** | **Primitive Specializations** |
| Predicate<T> | T->boolean | IntPredicate, LongPredicate, DoublePredicate |
| Consumer<T> | T->void | IntConsumer, LongConsumer, DoubleConsumer |
| Function<T,R> | T->R | IntFunction<R>,IntToDoubleFunction,IntToLongFunction,  LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T> |
| Supplier<T> | ()->T | BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier |
| UnaryOperator<T> | T->T | IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator |
| BinaryOperator<T> | (T,T)->T | IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator |
| BiPredicate<L,R> | (L,R)->boolean |  |
| BiConsumer<T,U> | (T,U)->void | ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsume<T> |
| BiFunction<T,U,R> | (T,U)->R | ToIntBiFunction<T,U>, ToLongBiFunction<T,U>,  ToDoubleBiFunction<T,U> |

**1.Predicate:**

In mathematical logic, a predicate is a function that receives a value and returns a boolean value.

@FunctionalInterface  
public interface Predicate<T>{  
 boolean test(T t);  
}

**test(T t) :** Evaluates this predicate on the given argument.

Parameters:T - the input argument

Returns: true if the input argument matches the predicate, otherwise false

The Predicate functional interface is a specialization of a Function that receives a generified value and returns a boolean. A typical use case of the Predicate lambda is to filter a collection of values:

List<String> names = Arrays.asList("Angela", "Aaron", "Bob", "Claire", "David");  
  
List<String> namesWithA = names.stream()  
 .filter(name -> name.startsWith("A"))  
 .collect(Collectors.toList());

In the code above, we filter a list using the Stream API and keep only the names that start with the letter “A”. The Predicate implementation encapsulates the filtering logic.

**Example2:**

public void practicePredicate()  
{  
 System.out.println("Predicate Example");  
 ArrayList<String> strings=new ArrayList<>();  
 strings.add("Predicate");  
 strings.add("");  
 strings.add("Function");  
 strings.add("");  
 strings.add("Example");  
 strings.add("");  
 List<Integer> integerList= List.of(1,2,3,4,5,6,7);  
 Predicate<String> isNotEmpty=s->!s.isEmpty();  
 Predicate<String> filter=s->s.contains("i");  
 Predicate<Integer> evenOrNot=e->(e&1)==0;  
 List<String> filteredList=strings.stream().filter(s->filter.test(s)).collect(Collectors.toList());  
 System.out.println(filterList(strings,isNotEmpty));  
 System.out.println(filteredList);  
 System.out.println(filterList(integerList,evenOrNot));  
}

**Generic Function to call predicate over list of values:**

private <T> List<T> filterList(List<T> list,Predicate<T> predicate)  
{  
 List<T> result=new ArrayList<>();  
 for(T item: list)  
 {  
 if(predicate.test(item))  
 {  
 result.add(item);  
 }  
 }  
 return result;  
}

**2.Consumer:**

The Consumer accepts a generified argument and returns nothing. It is a function that is representing side effects.

@FunctionalInterface  
public interface Consumer<T>{  
 void accept(T t);  
}

**Accept(T t):** This method accepts one value and performs the operation on the given argument  
**Syntax:**   
 void accept(T t)

**Parameters:** This method takes in one parameter:

For instance, let’s greet everybody in a list of names by printing the greeting in the console. The lambda passed to the List.forEach method implements the Consumer functional interface:

List<String> names = Arrays.asList("John", "Freddy", "Samuel");  
names.forEach(name -> System.out.println("Hello, " + name));

**Example2:**

public void practiceConsumer()  
{  
 System.out.println("Consumer Example");  
 List<Integer> list=List.of(12,34,56,71,82);  
 Consumer<Integer> consumer=e->System.out.println(e);  
 consumer.accept(2);  
 printElements(list,consumer);  
}  
private <T> void printElements(List<T> list,Consumer<T> consumer)  
{  
 for(T t:list)  
 {  
 consumer.accept(t);  
 }  
}

**3.Supplier:**

The Supplier functional interface is yet another Function specialization that does not take any arguments but produces a value of type T. We typically use it for lazy generation of values.

@FunctionalInterface  
public interface Supplier<T>{  
 T get();  
}

**T get():** This method does not take in any argument but produces a value of type T.

**Syntax:** T get()

For instance, let’s define a function that squares a double value. It will not receive a value itself, but a Supplier of this value:

public double squareLazy(Supplier<Double> lazyValue) {  
 return Math.pow(lazyValue.get(), 2);  
}

**Example2:**

public void practiceSupplier()  
{  
 System.out.println("Supplier Example");  
 Supplier<String> stringSupplier=()->new String("Example String");  
 System.out.println(stringSupplier.get());  
 Supplier<Double> doubleSupplier=()->Math.random();  
 System.out.println(doubleSupplier.get());  
}

**4.Function:**

The function type functional interface receives a single argument, processes it, and returns a value. One of the applications of this type of functional interface is taking the key from the user as input and searching for the value in the map for the given key.

@FunctionalInterface  
public interface Function<T, R>{  
 R apply(T t);  
}

**R apply(T t):**

**Parameters:** This method takes in only one parameter **t** which is the function argument

**Return Type:**This method returns the **function result** which is of type R.

**Example:**

public void practiceFunction()  
{  
 System.out.println("Function Example");  
 List<String> list=List.of("Kit","Kat","Shake");  
 Function<String,Integer> stringLengthFunction=e->e.length();  
 System.out.println(map(list,stringLengthFunction));  
}  
  
 private <T,R> List<R> map(List<T> list, Function<T, R> function) {  
 List<R> result=new ArrayList<>();  
 for(T e:list)  
 {  
 result.add(function.apply(e));  
 }  
 return result;  
 }

**5.UnaryOperator:**

A **UnaryOperator** **is like a function where it takes an argument and returns a result, but both the argument and the result are the same type.**

@FunctionalInterface  
public interface UnaryOperator<T> extends Function<T, T>{   
}

As UnaryOperator inherits from Function, it inherits apply() abstract function from Function Interface and this makes it Functional Interface.

**Example:**

private <T,R> List<R> map(List<T> list, Function<T, R> function) {  
 List<R> result=new ArrayList<>();  
 for(T e:list)  
 {  
 result.add(function.apply(e));  
 }  
 return result;  
}  
public void practiceUnaryOperator()  
{  
 System.out.println("Unary Operator Example");  
 List<Integer> list=List.of(10,20,30,40,50,60);  
 UnaryOperator<Integer> unaryOperator=i->i\*100;  
 System.out.println(map(list,unaryOperator));  
}

**6.BiFunction:**

The Bi-Function is substantially related to a Function. Besides, it takes two arguments, whereas Function accepts one argument.

@FunctionalInterface  
public interface BiFunction<T, U, R>  
{   
 R apply(T t, U u);  
}

In the above code of interface, T and U are the inputs, and there is only one output which is R.

**Example:**

public void practiceBiFunction()  
{  
 System.out.println("BiFunction Example");  
 BiFunction<String,String,Integer> biFunction=(a,b)->(a+b).length();  
 System.out.println(biFunction.apply("BiFunction","Example"));  
}

**7.BinaryOperator:**

Binary Operator takes two values and returns one value comparable to Bi- Function but similar to a Unary Operator, the input and output value types must be identical and of the same type.

In simple words, Unary Operator extends Function, and Binary Operator extends Bi-Function.

@FunctionalInterface  
public interface BinaryOperator<T> extends BiFunction<T, T, T>{  
}

**Example:**

public void practiceBinayOperator()  
{  
 System.out.println("BinaryOperator Example");  
 BinaryOperator<String> binaryOperator=(a,b)->a+"."+b;  
 System.out.println(binaryOperator.apply("BinaryOperator","Example"));  
}

**Method Reference:**

**Method references are a special type of lambda expressions**. They’re often used to create simple lambda expressions by referencing existing methods.

There are four kinds of method references:

* Static methods
* Instance methods of particular objects
* Instance methods of an arbitrary object of a particular type
* Constructor

**1) Reference to a static method:** Method reference to a static method of class.

If a Lambda expression is like:

// If a lambda expression just call a static method of a class   
(args) -> Class.staticMethod(args)

Then method reference is like:

// Shorthand if a lambda expression just call a static method of a class   
**Class::staticMethod**

public static void staticMethods()  
{  
 Supplier<String> stringSupplier=()->new String("A String");  
 System.out.println(stringSupplier.get());  
 Supplier<Double> randomNumber=Math::random;  
 System.out.println(randomNumber.get());  
}

**2)Reference to an instance method of a particular object:** Method reference to instance method of class.

If a Lambda expression is like:

// If a lambda expression just call a default method of an object

(args) -> obj.instanceMethod(args)

Then method reference is like:

// Shorthand if a lambda expression just call a default method of an object

**obj::instanceMethod**

public void instanceMethods()  
{  
 List<Integer> list=List.of(34,67,8,23,67,89,90);  
 Consumer<Integer> consumer=System.out::println;//static reference1  
 consumer.accept(56);  
 printElements(list,consumer);  
}  
private static <T> void printElements(List<T> list, Consumer<T> consumer) {  
 for(T t: list)  
 {  
 consumer.accept(t);  
 }  
}

**3) Reference to an instance method of an arbitrary object of a particular type:** Method reference to an instance method of an input object of a particular type.

If a Lambda expression is like:

// If a lambda expression just call an instance method of a ObjectType

(obj, args) -> obj.instanceMethod(args)

Then method reference is like:

// Shorthand if a lambda expression just call an instance method of a ObjectType

**ObjectType::instanceMethod**

public void methodReferenceObjectOfParticularType()  
{  
 List<String> list=List.*of*("Kit","kat","Shake");  
 Function<String,Integer> function=String::length;  
}

**4) Constructor method reference:**

If a Lambda expression is like:

// If a lambda expression just create an object   
(args) -> new ClassName(args)

Then method reference is like:

// Shorthand if a lambda expression just create an object   
ClassName::new

public void constructorReference()  
{  
 Function<Runnable,Thread> threadGenerator=Thread::new;  
 Runnable task=()->System.*out*.println("Thread execution");  
 Thread thread=threadGenerator.apply(task);  
 thread.start();  
 threadGenerator.apply(()->System.*out*.println("Thread execution2")).start();  
}

**Overall Example:**

import java.util.ArrayList;  
import java.util.Collections;  
import java.util.List;  
import java.util.Random;  
  
public class MethodReference2 {  
 List<Person> personList = new ArrayList<>();  
 MethodReference2()  
 {  
 personList.add(new Person("vicky", 24));  
 personList.add(new Person("poonam", 25));  
 personList.add(new Person("sachin", 19));  
 }  
 public static int compareByName(Person a,Person b)  
 {  
 return a.getName().compareTo(b.getName());  
 }  
 public static int compareByAge(Person a,Person b)  
 {  
 return a.getAge().compareTo(b.getAge());  
 }  
 public void methodRefernceStatic()  
 {  
 // Using static method reference to sort array by name  
 Collections.*sort*(personList, MethodReference2::*compareByName*);  
 // Using static method reference to sort array by age  
 Collections.*sort*(personList, MethodReference2::*compareByAge*);  
 }  
 public void methodReferenceInstance()  
 {  
 // A comparator class with multiple comparator methods  
 ComparisonProvider comparator = new ComparisonProvider();  
 // Using instance method reference to sort array by name  
 Collections.*sort*(personList, comparator::compareByName);  
 // Using instance method reference to sort array by age  
 Collections.*sort*(personList, comparator::compareByAge);  
 }  
 public void methodReferenceObjectOfParticularType()  
 {  
 List<String> personList2 = new ArrayList<>();  
 personList2.add("vicky");  
 personList2.add("poonam");  
 personList2.add("sachin");  
 // Method reference to String type  
 Collections.*sort*(personList2,String::compareToIgnoreCase);  
 personList2.forEach(System.*out*::println);  
 }  
 public void constructorReference()  
 {  
 Random ran = new Random();  
  
 // Assigning a random value to name  
 String name  
 = ran  
 .ints(97, 122 + 1)  
 .limit(7)  
 .collect(StringBuilder::new,  
 StringBuilder::appendCodePoint,  
 StringBuilder::append)  
 .toString();  
 }  
}

ComparisonProvider.java

public class ComparisonProvider {  
 public int compareByName(Person a, Person b)  
 {  
 return a.getName().compareTo(b.getName());  
 }  
 public int compareByAge(Person a, Person b)  
 {  
 return a.getAge().compareTo(b.getAge());  
 }  
}

Person.java

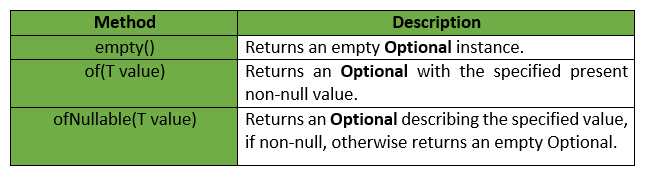
class Person {  
 private String name;  
 private Integer age;  
  
 public Person(String name, int age)  
 {  
 this.name = name;  
 this.age = age;  
 }  
  
 public Integer getAge() { return age; }  
 public String getName() { return name; }  
}

**Optional class:**

Every Java Programmer is familiar with **[NullPointerException](https://www.geeksforgeeks.org/null-pointer-exception-in-java/)**. It can crash your code. And it is very hard to avoid it without using too many null checks. So, to overcome this, Java 8 has introduced a new class Optional in **java.util package**. It can help in writing a neat code without using too many null checks. By using Optional, we can specify alternate values to return or alternate code to run.

**Creating Optional:**

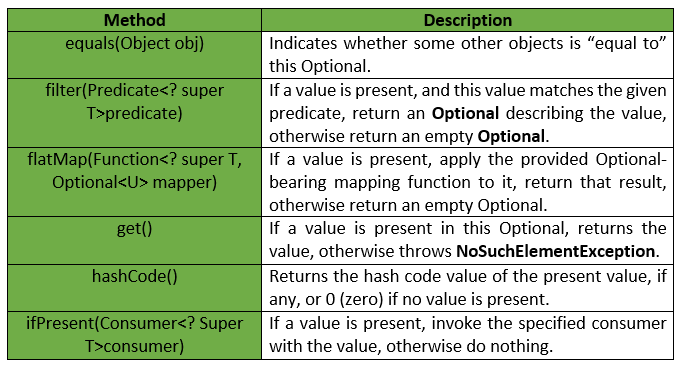
The following table shows the list of Static Methods provided by Optional Class :

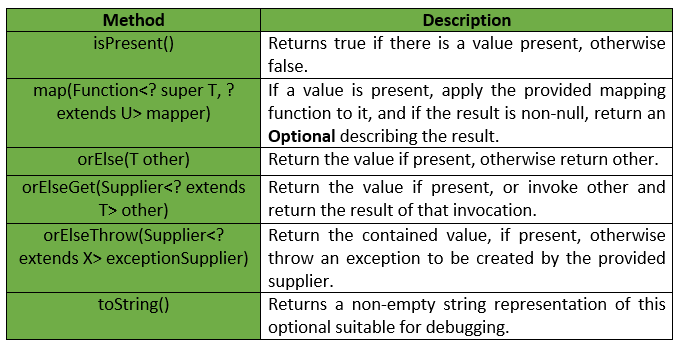


If we use null as argument to of() method, it throws exception. But ofNullable() method handles null value also.

void creatingOptional()  
{  
 System.*out*.println("Creating Optional ");  
 Optional<Person> optionalPerson1=Optional.*empty*();//empty optional  
 Person person=new Person("Praneetha",23);  
 Optional<Person> optionalPerson2=Optional.*of*(person);  
 Optional<Person> optionalPerson3=Optional.*ofNullable*(person);  
 Optional<Person> optionalPerson4=Optional.*ofNullable*(null);  
 // Optional<Person> optionalPerson3=Optional.of(null);  
 System.*out*.println(optionalPerson1);  
 System.*out*.println(optionalPerson2);  
 System.*out*.println(optionalPerson3);  
 System.*out*.println(optionalPerson4);  
}

The following table shows the list of Instance Methods provided by the Optional Class :



**Checking for Values and Doing Something if Value is Present Example:**

void checkingForValues()  
{  
 System.*out*.println("Checking for values using Optional Methods");  
 Optional<Person> optionalPerson1=Optional.*empty*();//empty optional  
 Person person=new Person("Praneetha",23);  
 Optional<Person> optionalPerson2=Optional.*of*(person);  
 Optional<Person> optionalPerson3=Optional.*ofNullable*(null);  
 if (optionalPerson1.isPresent()) {  
 System.*out*.println(optionalPerson1.get());  
 } else {  
 System.*out*.println("No person stored");  
 }  
 if (optionalPerson2.isPresent()) {  
 System.*out*.println(optionalPerson2.get());  
 } else {  
 System.*out*.println("No person stored");  
 }  
 optionalPerson2.ifPresent(System.*out*::println);  
 optionalPerson2.ifPresent(  
 p -> System.*out*.println("The name is " + p.getName())  
 );  
}

**Using Alternate Values if Value is not Present : orElse() and orElseGet() Example:**

void usingAlternateValues()  
{  
 System.*out*.println("Using alternate values using Optional Methods");  
 Optional<String> myString = Optional.*of*("My string");  
 Optional<String> noString = Optional.*empty*();  
 System.*out*.println(myString.orElse("Default String"));  
 System.*out*.println(noString.orElse("Default String"));  
 System.*out*.println(myString.orElseGet(()->"Default String"));  
 System.*out*.println(noString.orElseGet(()->"Default String"));  
 System.*out*.println(noString.orElseThrow(()->new RuntimeException()));  
}

orElse and orElseGet work similarly for returning an alternative (default) value if no value is present in the Optional object but has a subtle difference. The method provided in the orElse always gets executed no matter what value is stored in the Optional object. On the other hand, orElseGet accepts a lambda expression of the Supplier type and gets invoked if and only if the value is absent in the Optional object. Therefore, orElseGet performs much better than orElse.

**Optional Value Transformation With map() & flatMap():**

void mappingValuesWithOptionals()  
{  
 System.*out*.println("Transformation of optional values With map() & flatMap()");  
 Optional<String> myString = Optional.*of*("My string");  
 Optional<String> noString = Optional.*empty*();  
  
 System.*out*.println(myString.map(String::toLowerCase));  
 System.*out*.println(noString.map(String::toLowerCase));  
 Optional<Optional<String>> anotherOptionalString = Optional.*of*(  
 Optional.*of*("My another string")  
 );  
 System.*out*.println(  
 anotherOptionalString.map(  
 anotherString -> anotherString.map(String::toUpperCase)  
 )  
 );  
  
 System.*out*.println(  
 anotherOptionalString.flatMap(  
 anotherString -> anotherString.map(String::toLowerCase)  
 )  
 );  
}

**Output:**

Transformation of optional values With map() & flatMap()

Optional[my string]

Optional.empty

Optional[Optional[MY ANOTHER STRING]]

Optional[MY ANOTHER STRING]

The map returns the result of the computation (toLowerCase method) wrapped inside Optional if there's a value present. Otherwise, it returns an empty Optional. It works differently than flatMap, which is map transforms values only when they are unwrapped, whereas flatMap takes a wrapped value and unwraps it before transforming it.

**Conditionally Returning Values With a filter():**

void optionalFilter()  
{  
 System.*out*.println(" Conditionally Returning Values With a filter()");  
 Optional<Integer> value = Optional.*of*(23);  
 // to get an Optional<Integer> which filters even number  
 System.*out*.println(value.filter(x -> x % 2 == 0));  
 // to get an Optional<Integer> which filters odd number  
 System.*out*.println(value.filter(x -> x % 2 == 1));  
}

**Output:**

Conditionally Returning Values With a filter()

Optional.empty

Optional[23]

The Optional values are filtered using the filter method that accepts a lambda expression of type Predicate that will store the appropriate check. **If the Predicate condition returns true then the corresponding value wrapped inside the optional is returned otherwise, an empty Optional is returned.**

**Streams & Parallel Streams:**

 The stream API allows you to perform operations on collections without external iteration.

**Note:**

* If we want to represent a group of objects as a single entity, then we should go for [collection](https://www.geeksforgeeks.org/collections-in-java-2/).
* But if we want to process objects from the collection then we should go for streams.

**What does streams do?**

1. Streams take the data.
2. Do all the processing.
3. Return the data into the container the user wants or consume the data.

**Java Stream Features**

* **Functional-style operations:** Streams support functional-style operations like map, filter, and reduce, which enable developers to express data processing logic in a more declarative and readable way.
* **Pipelining:** Streams allow for chaining multiple operations together to form a pipeline, where the output of one operation becomes the input of the next. This enables efficient and compact code without the need for intermediate collections.
* **Lazy evaluation:** Streams employ lazy evaluation, meaning that intermediate operations are not executed until a terminal operation is invoked. This improves efficiency by avoiding unnecessary computation.
* **Parallel processing:** Streams can be processed in parallel, leveraging multi-core architectures to improve performance for large datasets. Parallel streams automatically divide the data into multiple chunks and process them concurrently.
* **Stream is not collection and does not contain any data:** Stream is indeed a fancy iterator which just takes the data from a source and process that only once. Stream does not hold any data but processes the elements one by one. Just like fluid is being passed through a pipe.  once you operate on a stream it becomes empty. so, stream is immutable and we can use it only once we neither can add anything to an existing stream nor we can remove anything from it the most we can do is to read elements from it once operate on each element and return a new stream

**Different Operations on Streams:**

Stream provides various operations that can be chained together to produce results. Stream operations can be classified into two types.

* Intermediate Operations
* Terminal Operations

**1. Intermediate Operations:**

**Intermediate operations return a stream as the output**, and intermediate operations are not executed until a terminal operation is invoked on the stream. This is called lazy evaluation, and it is discussed in detail in the later section (Lazy Evaluation).

**Characteristics of Intermediate Operations:**

* Methods are chained together.
* Intermediate operations transform a stream into another stream.
* It enables the concept of filtering where one method filters data and passes it to another method after processing.

**Some Intermediate stream operations:**

**i) filter():**

The filter() method returns a stream with the stream's elements that match the given predicate. Predicate is a functional interface in Java that accepts a single input and can return a boolean value.

**Example:**

static void filterDemo(){  
 System.*out*.println("Filter Example that filters even from a list of 1 to 5 elements");  
 final List<Integer> list = new ArrayList<>(Arrays.*asList*(1, 2, 3, 4, 5));  
  
 final List<Integer> ans = list.stream()  
 .filter(value -> value % 2 == 0)  
 .collect(Collectors.*toList*());  
  
 System.*out*.println(Arrays.*toString*(ans.toArray()));  
}

**ii)map():**

[The map() method](https://www.scaler.com/topics/java-stream-map/) returns a stream with the resultant elements after applying the given function on the stream elements.

**Syntax:**

<R> Stream<R> map(Function<? super T, ? extends R> mapper)

* <R> : It is a type parameter section. The <R> declares a type parameter named R. It signifies that the R could be any type (like String, Integer, etc.). It's used in a declaration that tells the compiler that R will be used as a placeholder for a type.
* Stream<R>: It is the return type of the map function. The function will return a Stream, where R could be any type. That is, the function may return Stream of String or Stream of Integer or anything else, depending upon what type R is.
* map: map is a function name belonging to the Stream interface. It's used for transformation. It applies a given function to the elements of the Stream and incorporates the output into a new Stream.
* Function<? super T, ? extends R> mapper: This is the argument to the map function. The argument is a Function interface, which is a part of Java's functional programming aspect.
* The Function interface has two type parameters, T and R:
* T is the type of the input to the function.
* R is the type of the result of the function.

The wildcards (? super T and ? extends R) mean that this function can accept:

* As an input a value of type T or any superclass of T, due to the use of ? super T.
* As an output a value that is of type R or any subclass of R, due to use of ? extends R.

So overall, the function takes a Function object, that can accept "any type, that is a superclass of T" and returns "any type, that is a subclass of R". The map function itself will return a stream of these transformed outputs. This transformation function is applied to each element of the stream 'in turn'.

**Example:**

static void mapDemo(){  
 System.*out*.println("Map Example to print 5 Table");  
 final IntStream stream= IntStream.*range*(1,11);  
 stream.peek(x->System.*out*.print("5\*"+x+"=")).map(x->x\*5).forEach(System.*out*::println);  
}

**iii)sorted():**

The sorted() method returns a stream with the elements of the stream sorted according to natural order or the provided Comparator.

**Example:**

static void sortedDemo(){  
 System.*out*.println("Sorted() example");  
 final List<Integer> list = new ArrayList<>(Arrays.*asList*(5, 1, 3, 4, 2));  
  
 System.*out*.println("Ascending Order");  
 list.stream().sorted()  
 .forEach(System.*out*::println);  
  
 System.*out*.println("\nDescending Order");  
 list.stream().sorted(Comparator.*reverseOrder*())  
 .forEach(System.*out*::println);  
}

**iv) distinct():**

This distinct() method returns a stream consisting of distinct elements of the stream (i.e.) it removes duplicate elements.

**Example:**

static void distinctDemo(){  
 System.*out*.println("distinct() example");  
 final List<Integer> list = new ArrayList<>(Arrays.*asList*(5, 1, 3, 4, 2,5,2,1,3,7,9,10));  
 list.stream().distinct().forEach(System.*out*::println);  
}

**v) peek():**

The peek() method returns a stream consisting of the elements of the stream after performing the provided action on each element. This is useful when we want to print values after each intermediate operation.

**Example:**

static void peekDemo(){  
 System.*out*.println("peek() example");  
 final List<Integer> list = new ArrayList<>(Arrays.*asList*(1, 2, 3, 4, 5));  
  
 final List<Integer> ans = list.stream()  
 .filter(value -> value % 2 == 0)  
 .peek(value -> System.*out*.println("Filtered " + value))  
 .map(value -> value \* 10)  
 .collect(Collectors.*toList*());  
}

**Output:**

Filtered 2

Filtered 4

**2. Terminal Operations:**

Terminal operations produce the results of the stream after all the intermediate operations are applied, and we can no longer use the stream once the terminal operation is performed.