Java8 Notes

25 October 2023 23:14

```
The main objective of Lambda expression:
To bring benefits of functional programming in Java


What is a Lambda expression?
----------------------------------------
It is an anonymous function -
without name,
without return type,
without modifiers.

Example1:
public void m1(){
 System.out.println("Hello");
}

() -> { System.out.println("Hello"); }
Note : If body contains a single line, then curly brackets are optional.

Example2:
public void m1(int a, int b){
     System.out.println(a+b);
}

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

Complier can guess the 'data type' automatically, we don't need to specify
'data type'.
(a,b) -> System.out.println(a+b);

Example3:
public int squareInt(int n){
     return n*n;
}

No name, no modifier, no return type
(int n) -> { return n*n; }

OR

n-> n*n;

Note: We don't need to specify 'return' without curly brackets  in case of
single line code.
Because compiler can able to guess it’s a return value.

Example4:
Public void m1(String s){
     Return s.length();
}

s -> s.length();
--------------------------------------------------------------------------------
----------------------------------------
```

FI ==> Functional Interface

Runnable ==> run()
Comparable ==> compareTo()
Comparator ==> compare()
ActionListener ==> actionPerformed()
Callable ==> call()

These all are interfaces. They have only one single abstract method. (SAM)

From 1.8v onwards, we can include 'default' and 'static' methods in normal interface.

Can we include 'default and static methods' in 'Functional Interface'?
The restriction is only applicable for 'abstract' methods. Functional interface should contain
exactly one abstract method but it have any number of default or static methods.

Example :

```
@FunctionalInterface
interface alph
{
     public void m1();

     default void m2(){
     ------------------
     }

      public static void m3(){
     -------------------
     }
}
```

To explicitly specify a functional interface, we can use an annotation - @FunctionalInterface.

Functional Interface wr.t Inheritance ----->

Example1:

```
@FunctionalInterface
interface A{
     public void m1();
}

@FunctionalInterface
interface B extends A
{
------------
}
```

Interface A is a valid Functional Interface.
Interface B is also a valid Functional Interface, because it extends parent class abstract method m1().

Example 2:

```
@FunctionalInterface
Interface A{
     Public void m1();
}
@FunctionalInterface
Interface A extends B{
     Public void m1();
}
```

Here, interface A is valid and Interface B is also valid because we're overriding the existing m1() method in interface B.
So, interface B has only one abstract method.

Example 3:

```
@FunctionalInterface
Interface A{
     Public void m1();
}
@FunctionalInterface
Interface A extends B{
     Public void m2();
}
```

Here, interface A is a valid interface, but interface is not valid FI because it has two abstract methods in
Total (m1 and m2 abstract method).

Example 4:

```
@FunctionalInterface
Interface A{
     Public void m1();
}
Interface A extends B{
     Public void m2();
}
```

Here A is a valid FI.
B is valid interface but it is not a valid FI.
So we can say @FunctionalInterface is an optional annotation.

Lambda expression with Functional Interfaces:

Example 1:

```
Interface Interf
{
     public void m1();
}

Class Demo implements Interf
{
     public void m1(){
     System.out.println("Hello…..");
     }
}
```

```
Class Test {
     public static void main(String args[]){

          //Using class Demo
          Demo d = new Demo();
          d.m1();

          //Using lambda expression and w/o Demo class
          Interf I = () -> System.out.println("Hello……");
          i.m1();


          // check whether it is valid or not
          Interf i =  new Demo();
          i.m1();
          ----it is valid because parent reference can be used to hold child
          object
     }
}
```

Here, in order to implement m1() method, we have created a top level class - Demo.
Instead of using a class, we can provide direct implementation using Lambda expression.

Example 2:

```
Interface Interf
{
     public void add(int a, int b);
}
Class Test {
     public static void main(String args[]){

          //Using lambda expression and w/o Demo class
          Interf I = (a,b) -> System.out.println("The sum = "+(a+b));
          i.add(5,6);
     }
}
```

Note: Lambda expressions are specific to Functional Interfaces. It cannot be used with generic interfaces.

Multithreading with Lambda expression

```
public class myRunnable implements Runnable {
    @Override
    public void run() {
        for(int i=0;i<10;i++)
            System.out.println("Child thread");
    }
}


public class test{
```

```
    public static void main(String[] args) {
        myRunnable r= new myRunnable();
        Thread th = new Thread(r);
        th.start();
        for(int i=0;i<10;i++)
        {
            System.out.println("Main thread");
        }
    }
}
```

Output will vary - main thread -- child thread OR child thread - main thread….
Benefit of Multithreading is Performance Improvement.

Note: Runnable is a functional interface so we can implement using Lambda expression.

Using Lambda expression

```
public class test{
    public static void main(String[] args) {
        Runnable r= () -> {
            for(int i=0;i<10;i++)
                System.out.println("child thread");
        };
        Thread th = new Thread(r);
        th.start();
        for(int i=0;i<10;i++)
        {
            System.out.println("Main thread");
        }
    }
}
```

Collections with Lambda expression

```
import java.util.ArrayList;
public class collections {

    public static void main(String[] args) {

        ArrayList<Integer> arrayList= new ArrayList<>();
        arrayList.add(10);
        arrayList.add(20);
        arrayList.add(25);
        arrayList.add(15);
        arrayList.add(50);
        System.out.println(arrayList);

        //sort the elements in list
        Collections.sort(l, new Mycomparator());
    }
}
```

Comparator Interface
Int compare (Object obj1, Object obj2)
 Returns -ve iff obj1 has to come before obj2

```
     Returns +ve iff obj1 has to come after obj2
     Returns 0 iff obj1 and obj2 are equal
```

Implementing compare() method

```java
import java.util.Comparator;
class Mycomparator implements Comparator<Integer>
{
    @Override
    public int compare(Integer i1, Integer i2) {

        // ternary operator
        return i1 < i2 ? -1 : i1 > i2 ? 1 : 0;

        OR
        // if-else-if
        if (i1 < i2)
            return -1;
        else if(i1 > i2)
            return 1;
        else
            return 0;
    }
}
```

Is Comparator a 'functional interface' ?
Yes, it has only one abstract method. So, Comparator is a Functional Interface.

Using Lambda expression

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
public class collections {

    public static void main(String[] args) {

        ArrayList<Integer> arrayList= new ArrayList<>();
        arrayList.add(10);
        arrayList.add(20);
        arrayList.add(25);
        arrayList.add(15);
        arrayList.add(50);
        System.out.println(arrayList);

        // curly brackets and return are optional for single line code
        Comparator<Integer> c = (i1,i2) -> {
            return i1<i2? -1 : i1>i2 ? 1 : 0;
        };

        OR

        Comparator<Integer> c = (i1,i2) -> i1<i2 ? -1 : i1>i2 ? 1 : 0;

        Collections.sort(arrayList, c);
        System.out.println(arrayList);
```

```
    }
}
```

Anonymous Inner class vs Lambda Expression

Anonymous Inner class -
Can extend a normal class
Can extend an abstract class
Can implement an interface which contains any number of abstract methods

Lambda expression can implement an interface which contains a single abstract method (FI)
Lambda expression applicable only for Functional Interface

Anonymous Inner class != Lambda expression
Anonymous Inner class > Lambda expression

Default Methods in Interface

They are also known as Virtual Extension Method or Defender Method.

Interface -------

```java
package default_method;
/**
 * Interface1
 */
public interface Interface1 {
    public void m1();
    public void m2();
}
```

Class test1 ---------

```java
package default_method;
/**
 * test1
 */
public class test1 implements Interface1{
    @Override
    public void m1() {
        System.out.println("m1 implemented");
    }
    @Override
    public void m2() {
        System.out.println("m2 implemented");
    }

    public static void main(String[] args) {
        test1 ob = new test1();
        ob.m1();
        ob.m2();
    }

}
```

'
'
'
'
'
'
'
'
Class test100 ------------

```java
package default_method;
/**
 * test100
 */
public class test100 implements Interface1{
    @Override
    public void m1() {
        System.out.println("m1 implemented");
    }
    @Override
    public void m2() {
        System.out.println("m2 implemented");
    }

    public static void main(String[] args) {
        test100 ob = new test1();
        ob.m1();
        ob.m2();
    }

}
```

In the above class test1 --- class test100, we have implemented two abstract methods of interface Interface1.

Suppose, we need to add 1 more abstract method(m3) in interface Interface1.

In order to do so, we need to modify all the classes 1- 100, and provide implementation for the new abstract method m3.

So, to solve this problem. Java8 introduced 'default methods' in Interface.

Without affecting the implementing classes if we want to add new methods in interface => Default methods

```java
public interface Interface1 {
     default void m3(){
        System.out.println("Default methods in Interfacer");
    }
}
```

Default methods will have a default implementation ,i.e. it is not abstract method.

So implementing classes don't need to provide an implementation to default method.

Note : Default methods can be overridden in implementing classes if required.

```
public class test100 implements Interface1{

    @Override
    public void m3(){
        System.out.println("Overriding default implementation of default
methods");
    }

    public static void main(String[] args) {
        test100 ob = new test1();
        ob.m3();
    }

}
```

Note: We can't override Object class methods in interface.
Example:

```
interface I {
     default int hashcode(){
          Return 10;
}
```

Error : default method hashCode in interface I overrides a member of java.lang.Object default int hashCode().

Note: It is not allowed to define Object class methods as default methods in interface.

Does Java support multiple inheritance?

No, Java doesn't support multiple inheritance.

Reason :

P1 class - m1 method
P2 class - m1 method

P3 class extends P1, P2 --- then there will be an ambiguity for m1 method
This problem is also known as 'Diamond problem'.

But, Python supports multiple inheritance. How?

Let's see:

Python code:

```
     class p1:
          def m1(self):
               print('P1 method')

     class p2:
          def m1(self):
               print ('P2 method')
```

```
     class C(p1,p2):pass

      c = C()
      c.m1()
```

```
Result :
P1 method

Python supports multiple inheritance as it consider the order of classes are
extended.
- P1 -> P2

i.e. it checks whether P1 class has a method 'm1' then it checks whether p2
class contains method 'm1'.

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

Diamond Problem in Default methods in interface

If a class implements multiple interfaces, having same default method,
which default method implementation will be considered by implementing class?
```

```
interface Left{
     default void m1(){
     System.out.println("Left interface m1 method");
     }
}

interface Right {
     default void m1(){
     System.out.println("right interface m1 method");
     }
}

class Test implements Left, Right{

}
```

```
In this case, java compiler will throw an error.
Error: class Test inherits unrelated defaults for m1() from typesLeft and
Right

Solution :
```

Scenario 1 : We will override(provide our own impl.) the default methods in the implementing class Test

```
class Test implements Left, Right{

     public void m1(){
          System.out.println("Our own m1 method");
     }

     Public static void main(String args[]){
```

```
          Test t =  new Test();
          t.m1();
     }
}
```

Output: Our own m1 method

Scenario 2: We want implementation of Left interface to be executed.

We'll use this command -> '**<interface_name>.super.<method_name>' inside m1 method**

```
class Test implements Left, Right{

     public void m1(){
          // System.out.println("Our own m1 method");

          Left.super.m1();
     }

     Public static void main(String args[]){
          Test t =  new Test();
          t.m1();
     }
}
```

Output: Left interface m1 method

Which is more costly 'class' or 'interface'?

Class is more costly than interface.

Because interface never contain constructor, static block, instance variable, etc.
i.e. interface is a light-weight component.

Note: We never create an object for interface.

Suppose we have created a static method mstat()

```
     public static void m1(){}
```

As we know, we don't need an object to execute static method.

Why we do create 'static method' in a 'class', instead we should go for 'interface ' ?

Interface is light and cheaper than classes.

we don't need an object for static methods.

It's better to go for an interface.

Therefore, from Java 1.8v, static methods were introduced in Java.

Static methods in Interface

```
package default_method;
public interface Interface2 {

    public static void m1(){
        System.out.println("Static methods in interface");
    }
}
```

Implementing class test2

```
package default_method;
public class test2 implements Interface2{
    public static void main(String[] args) {



     // calling static methods of interface
        Interface2.m1();

    // invalid
        m1();
        test2.m1();
       test2 ob = new test2();
       ob.m1();

    }
}
```

Output - Static methods in interface

How to call static methods in Implementing class?

Command - **<interface_name>.<static_method_name>();**

Note: Static methods of an interface are not available for implementing classes. We can call 'static_methods' using 'interface_name' only.

Is it possible to have 'main method' in interface?

Yes, from Java 1.8v onwards, we can create static and main methods in interface.

```
public interface Interface2 {

    public static void m1(){
        System.out.println("Static methods in interface");
    }
    public static void main(String[] args) {
```

```
        m1();
    }
}
```

Note: If everything is static, then don't go for classes, instead use 'Interfaces'.

Predefined Functional Interfaces

- Predicate - when we want result type as 'boolean'.
- Function - when we want to perform some operation and get some result.
- Consumer
- Supplier

Two argument Predefined functional interface

- BiPredicate
- BiFunction
- BiConsumer

Primitive Functional interface

- IntPredicate
- IntFunction
- IntConsumer

Predicate FI :

It contains only one abstract method

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

Let's suppose we provide an implementation to the method --

```
public boolean test(Integer i){
     if (i%2==0)
          return true;
     else
          return false;
}
```

Equivalent Lambda expression for above method : i -> i%2==0
(For single line code - brackets are optional)

This will return a boolean value.

Note: whenever we need a boolean value as a result, we can use Predicate as functional interface.

Let' see :

```
import java.util.function.Predicate;
/**
 * predicate
 */
```

```
public interface checkEven {

    public static void main(String[] args) {

        Predicate<Integer> p1 = i -> i%2==0;
        System.out.println(p1.test(10));
        System.out.println(p1.test(15));

    }
}

Output:
True
False
```

Note: In order to use Predicate FI, we use import statement
`import java.util.function.Predicate`

Optional: We have used 'interface' instead of 'class'. Because we're just using a single static method (no objects).

Requirements:

1. Whenever we need to apply multiple conditions: Use Predicate FI

```
Predicate<Employee> p1 = e->e.salary>10000 &&
e.isHavingGf==true---------------;
```

2. Write a Predicate to check whether length of string is > 5 or not

```
Predicate<String> p1 = s -> s.length() > 5;
System.out.println(p1.test("ALPHA");
System.out.println(p1.test("BOX");
```

3. Write a predicate to print array of string whose string is > 5.

```
String s[] = {"Alpha", "Mangoes", "Butterfly", "Kite"};
Predicate<String> p1 = s -> s.length() > 5;

//print all string whose length > 5
for (String str : s)
{
     if (p1.test(str))
          System.out.println(str);
}
Output:
Mangoes
Butterfly
```

4. Write a predicate to print string whose length is even.

```
String s[] = {"Alpha", "Mangoes", "Butterfly", "Kite"};
Predicate<String> p1 = s->s.length()%2==0;
for (String str :s)
{
     if (p1.test(str))
     System.out.println(str);
}
```

Output:
Kite

5. Create a class employee and return list of employees whose salary is greater than 50,000 using Predicate FI.

```
public class employee {

    String name;
    int salary;

    employee(String name, int salary){
        this.name= name;
        this.salary = salary;
    }
}
```

```
import java.util.ArrayList;
import java.util.function.Predicate;
public class main {

    public static void main(String[] args) {

        ArrayList<employee> emplist = new ArrayList<>();
        emplist.add(new employee("arjun", 20000));
        emplist.add(new employee("mahesh", 40000));
        emplist.add(new employee("kartik", 60000));
        emplist.add(new employee("vishnu", 100000));

        Predicate<employee> p1 = e -> e.salary > 50000;

        for (employee e : emplist) {
            if (p1.test(e))
                System.out.println(e.name + " " + e.salary);
        }
    }
}
```

Output:
Kartik
Vishnu

Predicate Joining (AND, OR , Negate) - Merge two or more predicates

p1 -> to check whether the number is even or not
p2 -> to check whether the number is > 10 or not

And - p1.and(p2).test(34)

```
import java.util.function.Predicate;
public class joinPredicate {

    public static void main(String[] args) {

        int arr [] = {2, 3, 4, 5, 6, 7, 8};
```

```
        Predicate<Integer> p1 = i->i%2==0;

        Predicate<Integer> p2 = i->i>5;

        for(int value : arr){

            if(p1.and(p2).test(value))
                System.out.println(value);

        }
    }
}
Output:
6
8
```

OR - p1.or(p2).test(34)

```
import java.util.function.Predicate;
public class joinPredicate {

    public static void main(String[] args) {

        int arr [] = {2, 3, 4, 5, 6, 7, 8};

        Predicate<Integer> p1 = i->i%2==0;

        Predicate<Integer> p2 = i->i>5;

        for(int value : arr){

            if(p1.or(p2).test(value))
                System.out.println(value);

        }
    }
}
Output:
2
4
6
7
8
```

NOT - p1.negate().test(23)

```
import java.util.function.Predicate;
public class joinPredicate {

    public static void main(String[] args) {

        int arr [] = {2, 3, 4, 5, 6, 7, 8};

        Predicate<Integer> p1 = i->i%2==0;

        Predicate<Integer> p2 = i->i>5;

        for(int value : arr){

            if(p1.negate().test(value))
```

```
                System.out.println(value);

        }
    }
}
Output:
3
5
7
```

Note: Return type of 'Predicate' is always boolean.

Function Predefined FI
Whenever we want to perform some operation and expects some result.

```
Input->Perform operation->output
4    ->Square operation -> 16

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

Import statement - import java.util.function.Function;

Examples:

1. square of a number

```
Function<Integer,Integer> f1 = i->i*i;
System.out.println(f1.apply(5));
```

2. length of a string

```
Function<String,Integer> f2= s->s.length();
System.out.println(f2.apply("Aeroplane"));
```

3. string to UpperCASE

```
Function <String,String> f3 = s->s.toUpperCase();
System.out.println(f3.apply("Workspace"));
```

4. Grades for a student in an exam
    * >=80 - A
    * >=60 - B
    * >=50 - C
    * >=40 - D
    * <40 - E

Student class

```
public class Student {

    String name;
```

```
    int marks;
    Student(String name, int marks){
        this.name = name;
        this.marks = marks;
    }

}
```

Main class

```
Function<Student,Character> f4 = s->s.marks>=80 ? 'A' : s.marks>=60 ?
'B' : s.marks>=50 ? 'C' : s.marks>=40 ? 'D' : 'E';

Student s[]= {
            new Student("Sameer", 70),
            new Student("Rakesh", 80),
            new Student("Aman", 50),
            new Student("Puneet", 35)
        };

 char grade =' ';
 for(Student s1 : s){
    grade = f4.apply(s1);
    System.out.println("Student Name : "+s1.name);
    System.out.println("Student Marks : "+s1.marks);
    System.out.println("Student Grade : "+grade);
    System.out.println();
  }
```

5. Print student names whose marks are greater than 60 **using Predicate**.

```
  Predicate<Student> p1 = str->str.marks>=60;
  for(Student s1 : s){
       if (p1.test(s1)){
           System.out.println("Student Name : "+s1.name);
           System.out.println("Student Marks : "+s1.marks);
           System.out.println("Student Grade : "+f4.apply(s1));
           System.out.println();
       }
   }
```

Function Chaining

1. andThen - f1.andThen(f2).apply(i)
2. compose - f1.compose(f2).apply(i) - first f2 and then f1

```
Function<Integer,Integer> f5 = i->i*2;
Function<Integer,Integer> f6 = n->n*n*n;
System.out.println("And then : "+f5.andThen(f6).apply(2));
System.out.println("Compose : "+f5.compose(f6).apply(2));

Output:
And then : 64
Compose : 16
```

Consumer Predefined Functional Interface (void or no return type)
It will not return anything. It will just consumes the value.

Predicate<T> ---- boolean
Function<T,R> --- R value
Consumer<T> ----- void (it always print the value)

```
@FunctionalInterface
interface Consumer<T>{

    public void accept(T t);
}
```

Import statement - import java.util.function.Consumer;

Examples:

1. Write a consumer to print a message.

```
public class consumerFI {
    public static void main(String[] args) {

        Consumer<String> c1 = s->System.err.println(s);
        c1.accept("Welcome on board");
    }
}
```

2. Write a consumer to print the student details.

```
Student arr[] = {
                new Student("Ajay", 40),
                new Student("Vikram", 60),
                new Student("William", 80),
                new Student("John", 100)
        };

Consumer<Student> c2 = s -> System.out.println(s.name + " " + s.marks);

for (Student s1 : arr)
{
   c2.accept(s1);
}
```

3. Combination of 'Predicate', 'Function' and 'Consumer' functional programming.

```
Student arr[] = {
   new Student("Ajay", 40),
   new Student("Vikram", 60),
   new Student("William", 80),
   new Student("John", 100)
};

Function<Student,Character> f4 = s->s.marks>=80 ? 'A' : s.marks>=60 ?
'B' : s.marks>=50 ? 'C' : s.marks>=40 ? 'D' : 'E';

Predicate<Student> p1 = s->s.marks>=60;
```

```
Consumer<Student> c2 = s -> System.out.println("Student  Name : "+s.name
+ "Marks : " + s.marks+"Grade : "+f4.apply(s));

 for (Student s1 : arr) {
    if (p1.test(s1))
          c2.accept(s1);
 }
```

Supplier Predefined Functional Interface <R>

Just supply my required objects and it won't take any input.

```
@FunctionalInterface
interface Supplier<R> {

     public R get();

}
```

Import statement - import java.util.function.Supplier;

1. Print the system date using Supplier FI

```
import java.util.Date;
import java.util.function.Supplier;
public class supplierFI {

    public static void main(String[] args) {

        Supplier<Date> s1 = ()->new Date();
        System.out.println(s1.get());
    }
}
```

Output :
Sun Oct 29 12:44:40 IST 2023

2. Generate random OTP using supplier FI.

```
    Supplier<String> s2 = () -> {

        String otp ="";

        for(int i=0;i<6;i++)
        {
           otp = otp + (int)(Math.random()*10);
        }

        return otp;
    };

     System.out.println(s2.get());
     System.out.println(s2.get());
     System.out.println(s2.get());
     System.out.println(s2.get());
```

```
Output:
170022
125374
514875
697362
```

Two input arguments

```
BiPredicate Predefined Functional Interface <T1, T2>

Normal Predicate can take only one output and perform conditional checks.
Sometimes our programming requirement is we have to take 2 input arguments and
perform conditional checks. for this requirement we need to go for
BiPredicate.

BiPredicte is exactly same as Predicate but it takes two input arguments.

interface BiPredicate <T1,T2>
{
     public boolean test(T1 t1, T2 t2);
}
```

Import statement - import java.util.function.BiPredicate

1. Check whether the sum of two given integers is even or not.

```
BiPredicate<Integer,Integer> b1 = (a,b) -> (a+b)%2==0;

System.out.println(b1.test(2, 1));
```

```
Output:
False
```

```
BiFunction Predefined Functional Interface <T1, T2>
```

It is exactly same as Function Predefined Functional Interface but it takes two input parameters.

Import Statement - import java.util.function.BiFunction

1. Add employee objects to the Employee List using BiFunction

```
ArrayList<Employee> emp = new ArrayList<>();

BiFunction<Integer,String,Employee> bf = (eno, name) -> new Employee(eno,
name);

emp.add(bf.apply(101, "Smith"));
emp.add(bf.apply(102, "John"));
emp.add(bf.apply(103, "Nancy"));
emp.add(bf.apply(104, "Jay"));

for(Employee e : emp)
```

```
    {
        System.out.println(e.eno+" "+e.name);
    }
```

Output:
101 Smith
102 John
103 Nancy
104 Jay

BiConsumer Predefined Functional Interface <T1, T2>

It is exactly same as Consumer Predefined FI but it contains two input parameters.

It returns no value or it has void return type.

```
@FunctionalInterface
interface BiConsumer <T1. T2>
{
    public void accept(T1 t1. T2 t2);
}
```

1. Increase salary of all employees by x amount.

```
    ArrayList<employee> emplist = new ArrayList<>();

    BiConsumer<employee,Integer> bc = (e,x) -> e.salary = e.salary+x;

       emplist.add(new employee("Rakesh", 40000));
       emplist.add(new employee("Smith", 60000));
       emplist.add(new employee("John", 50000));
       emplist.add(new employee("Jack", 70000));

     for (employee e : emplist)
     {
           bc.accept(e, 500);

           System.out.println(e.name+" "+e.salary);
     }
```

Output:
Rakesh 40500
Smith 60500
John 50500
Jack 70500
