Table of Contents

Lambda Expressions

Reference:- http://www.java2novice.com/java-8/lambda-expression/

- lambda expressions brought functional programming feature to java.
- lambda expression is also called anonymous function without any declarations.
- anonymous function is a function with no return type, modifiers and without name.
- below are some examples of how to define lambda expression/functions.
- A lambda expression is an instance of a functional interface
- Examples

Syntax :- (arguments) -> {body}

here arguments can be of any type similar as we pass to any method.

examples

```
1. (i)-> {System.out.println(i);}
2. (int i) {System.out.println(i);}
3. ()-> {System.out.println("NoArgs");}
4. (int i,int j)-> {System.out.println("sum:- "+i+j);}
5. i-> {System.out.println(i);}
```

- body can contain any number of statements.

```
1) (int i, int j) ->  return i+j
2) (int i, int j) -> {
                         int i = i+j;
                        System.out.println(i);
                       }
```

- in body if you don't specify return type , void will be default return type.
- Examples

e.g.1

```
public void m1()
{
        Sysout("hello world..!")
}
```

lambda expression

```
()->{ Sysout("hello world..!") }
()->Sysout("hello world..!")   // curly braces are also optional when code not more than 1 line.
```

e.g.2

```
public void m1(int a,int b)
{
        sysout(a+b)
}

(int a,int b)->{ sysout(a+b) } or
(int a,int b)-> sysout(a+b)  or
(a,b) -> sysout(a+b)
```

e.g.3

```
public int squareIt(int n)
{
        return n*n;
}

(int n) -> {return n*n;} or // here return is required.
(int n) -> return n*n;  // here return is required.or
(int n) -> n*n;  // here return is not required.
(n) -> n*n; // or
n->n*n;   // if only one param , paranthesis is also optional.
```

e.g. 4

```
public int m1(String s1)
{
        return s1.length();
}

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

- to call these lambda expressions we use FI(functional interface.)
- FI :- are interface with one and only one abstract method.
- Functional interface can have static and default methods.

Functional interface

- An interface with only one abstract method is called functional interface
- in Java 8, we declare @FunctionalInterface annotation above the interface to notify java compiler that interface is FI.

e.g. 1

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

}

@FunctionalInterface
interface B extends A
{
        public void m1();  //overridden method in B. if we don't declare this method
then also it is inherited from A interface.
        public void m2();     // it is abstract method in Interface B. so functional
interface can have only 1 abstract method so we need to remove
@FunctionalInterface annotation form top.
}
```

e.g. 2

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

Class B implements A
{
        public void m1()
        { sysout("hello world..!"); }
}

Class Test
{
        psvm(---)
        {
                B b=new B();
                b.m1();
                //or
                A a=new B();
                a.m1();

                //or lambda expression will be like
                A a= ()-> Sysout("hello world...!");

        }

}
```

- so from above point it concludes like we use lambda expression along with functional interface.

//Threading example old style...

```
package test;

class ChildThreadClass implements Runnable
{
	public void run()
	{
		for(int i=0;i<10;i++)
		{
			System.out.println("ChildThread:- "+i);
		}
	}
}

public class LambdaThreadExample {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		Runnable r=new ChildThreadClass();

		Thread t=new Thread(r);
		t.start();

		for(int i=0;i<10;i++)
		{
			try
			{
				Thread.currentThread().sleep(1000);
			}
			catch(Exception e)
			{
			System.out.println("MainThread :- interrupted my sleep...!");
			}
				System.out.println("MainThread:- "+i);
		}
	}
}
```

//Threading example java8 style...

Multithreading example using lambda expression and FI

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

        // TODO Auto-generated method stub

        // old style
        //Runnable r=new ChildThreadClass();

        Runnable r=()->{

                for(int i=0;i<10;i++)
                {

                try
                {
                        Thread.currentThread().sleep(1000);
                }
                catch(Exception e)
                {
                        System.out.println("interrupted my sleep...!");
                }

                        System.out.println("ChildThread:- "+i);
                }
         };


                Thread t=new Thread(r);
                t.start();

                for(int i=0;i<10;i++)
                {

                        System.out.println("MainThread:- "+i);
                }
                }
        }
```

- Marker Interface is one which is having no/zero abstract method in it.
- Java API has many single method implementations (i.e. functional interfaces) like Comparator, Runnable, Callable,Consumer (argument of forEach() method.) etc.
- Prior to Java-8, you can implement and initiaize them using anynomus class syntax.
- Example usage of lambda Expression for Comparator.

```
Person.java
------------
package test;
public class Person {

        int id;
        String name;
        //Convenience constructor.
        public Person(int id, String name) {
                super();
                this.id = id;
                this.name = name;
        }
        //default constructor.
        public Person() {
                // TODO Auto-generated constructor stub
        }

        @Override
        public String toString() {
                return "Person [id=" + id + ", name=" + name + "]";
        }
        //getters and setters.
}


package test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class TestClass {

        public static void main(String[] args) {

                Person p=new Person(1,"suraj");
                Person p2=new Person(3,"Pankaj");
                Person p1=new Person (2,"Chetan");
                List<Person> l=new ArrayList();
                l.add(p); l.add(p2); l.add(p1);
                System.out.println(l.toString());

                //Lambda expression used here.
                Comparator employeeIdComparator=(Object obj1,Object obj2)->{

                        Person firstPerson=(Person)obj1;
                        Person secondPerson=(Person)obj2;

                        if(firstPerson.getId()>secondPerson.getId()) {
                                return 1;
                        }
                        else if(firstPerson.getId()<secondPerson.getId()) {
                                return -1;
                        }
                        else {
                                return 0;
                        }


                        };
                        Collections.sort(l, employeeIdComparator);
                        System.out.println(l.toString());
        }
}
```

Output :-

```
==========
[Person [id=1, name=suraj], Person [id=3, name=Pankaj], Person [id=2,
name=Chetan]]
[Person [id=1, name=suraj], Person [id=2, name=Chetan], Person [id=3,
name=Pankaj]]
```

- lambda expressions used in Comparator to sort a collection.

```
package test;

import java.util.*;


public class LambdaCollectionSortExample {

        public static void main(String[] args) {

                ArrayList<Integer> al=new ArrayList<Integer>();

                al.add(60);al.add(20);al.add(40);al.add(30);al.add(50);

                System.out.println("list as is :- "+al.toString());

                Comparator<Integer> comparatorObj=(input1,input2)->

                                //before uncommenting below add curly braces.
                                /*if(input1<input2)
                                {
                                        return -1;
                                }
                                else if(input1>input2)
                                {
                                        return 1;
                                }
                                else
                                { return 0;}*/
                        //if we write below line curly braces removed.from this lambda
                        //ascending order.
                        //(input1<input2)?-1:(input1>input2)?1:0;
                        (input1>input2)?-1:(input1<input2)?1:0;

                        Collections.sort(al,comparatorObj);
                        System.out.println("sorted list is :- "+al.toString());
        }
}
```

Lambda vs Annonymous Inner Classes.

- Annonymous inner classed can be replaced by Lambdas only when there is only one abstract method in that Anonymous inner class..
- Annonymous inner class are more powerful because consider an interface where we have more that 1 method, there we cannot create lambda expr.

```
Example 1

interface A
{
        public void m1();
        public void m2();
}

//annonymous innter class.
A a=new A()
{
        public void m1()
        {
                Sysout("m1");
        }

        public void m2()
        {
                Sysout("m2");
        }

}


a.m1()
a.m2()
```

```
Example 2

//annonymous inner class.
Runnable r=new Runnable(){
        public void run()
        {

                sysout("child thread from annonymous inner class.")
        }
}

//lambda expression for same.

Runnable r=()->sysout("child thread from annonymous inner class.")
```

Default Methods. / Virtual Extension Method / Defender Method.

- Default methods are added to java8 so that we can extend functionality of existing interface by adding some additional methods to it.
- if we try to add new methods to existing interfaces in java7, then all the implementation classes of java-7 would fail compiling since those classes are old and are not aware of new methods.
- such new methods added to existing interfaces are declared with default keyword and its implementation.
- default keyword we write at method declaration in interface only. overridden methods we don't declare default.
- Object class methods are not allowed to be made default in interface. because Object is already superclass for all java classes.
- in cases where there is conflict between interfaces, if both interfaces have definition of same method then there would be multiple inheritance case.in such cases we need to override that conflicting method and from that call whichever interface method we want to call.
- With default methods you can include method body within the interface.
- with 'default' keyword we can declare default methods in a interface.

```
public interface DemoInterface {

   default void greetingsMessage(){
      System.out.println("Welcome...");
   }
}
```

- with default methods if our class is not implementing the method then default method will be called. so no need of making our class as abstract.
- once overridden the method by the implementation class then we don't need to mention the method as default.
- Why do we need to implement a method within the interface?
- Let's say you have an interface which has multiple methods, and multiple classes are implementing this interface. One of the method implementations can be common across the class, we can make that method as a default method, so that the implementation is common for all classes.
- Second scenario where you have already existing application, for a new requirement we have to add a method to the existing interface. If we add new method, then we need to implement it throughout the implementation classes. By using the Java 8 default method we can add a default implementation of that method which resolves the problem.
- java was not having multiple inheritance. due to which there could be ambiguity of calling the method with same name which is present in two classes.
- similarly with default methods in two interfaces with same name and different or same body, there could be ambiguity while invoking the method.
- hence that default methods from two different interfaces become abstract for the implementation class. we have to implement that abstract method in our implementation class.

- similarly, if two interfaces are having same default method. and one is extending from other. then default method in second interface would be overridden.

- we can call default method of interface explicitly from our implementation class as :- InterfaceName.super.method();
- default methods made API's backword compatible.
- there can be 'n' number of default methods in the interface.
- one example of default method is ForEach() method provided in Iterable interface. this method is used to iterate over collection framework.

ForEach() method

- this method is provided to iterate over collection framework.
- It is defined in Iterable and Stream interface.
- Collection classes that uses Iterable interface have ForEach method.

```
        Example
        public class TestClass {

public static void main(String[] args) {

Map<String,String> vehicleCompany=new HashMap<String,String>();

        vehicleCompany.put("Tata", "Harrier");
        vehicleCompany.put("BMW", "X1");
        vehicleCompany.put("Mahindra", "XUV500");
        TestClass.iterate(vehicleCompany);

}

private static void iterate(Map vehicleCompany) {

        vehicleCompany.forEach((k,v)->{System.out.println("key:-"+k+"
        value:-"+v );});

}
```

- this default method heading in Map interface looks like :- default void forEach(BiConsumer<? super K, ? super V> action) {....}
- here BiConsumer is a funtional interface Represents an operation that accepts two input arguments and returns no result.
- BiConsumer is specialized form of Consumer(I).

```
        Example for iterating a List would be
        public class TestClass {

                public static void main(String[] args) {

                        List <String> vehicleList=new ArrayList<String>();
                        vehicleList.add("Harrier");
                        vehicleList.add("X1");
                        vehicleList.add("XUV500");

                        TestClass.iterate(vehicleList);
                }

                private static void iterate(List l) {
                        l.forEach(item->{System.out.println(item);});
                }
        }
```

- ForEach default method heading in List interface looks like :-

```
    default void forEach(Consumer<? super T> action) {
```

- Here Consumer is functional interface. it represents an operation that accepts a single input argument and returns no result.
- ForEach has it's own default method implementation for 1-D(List) & 2-D(Map) interfaces.
- major difference beteween iterating using for loop in earlier java versions is we were using external loop.and from java 1.8 onwards that loop is provided internally. so we can use that loop insted of writing our own loop.

Method and Constructor Reference

- both serve purpose of code reusability.
- instead of writing a lambda expression and call a method, we can use Method reference.
- this is short hand way of writing lambda expression.
- it is implicit way of lambda expression. compiler will write lambda expression for us when we use method reference.
- What is need of method reference ? :- this is short hand way of writing lambda expression.
- There are three types of Method Reference

    - Static method reference.
    - instance method reference.
    - constructor method reference.

Method Reference

- Functional Interface object can be assigned to already exisitng specified Method with :: operator is Method referance.
- both Static and Instance method are allowed to be called with reference symbol ::
- Static Method can be called wrt classname Test::m1();
- Instance Method can be called wrt instance t::m1();

  e.g. Test t=new Test();

- in method reference , arguments passed must match.
- Method Reference is alternative to lambda expression.

Constructor Reference

- ClassName::new e.g. Test:: new   --------> this is constructor referance.
- Operator used for method reference :: (method reference operator)

- Consider below Example. in same example we can observe instance method reference by making printMessage() method of HelloMessagePrinter class as instance and uncommenting the code with comment instance method reference
- thing to note here is we can do same thing with help of lambda expression.

```
@FunctionalInterface

interface MessagePrinter {

        void printMessage();

}

class HelloMessagePrinter {

        public static void printMessage(){

                System.out.println("Hello World...!");

        }

}

public class Executor {

        public static void main(String[] args) {

                //static method reference.
                MessagePrinter mp= HelloMessagePrinter::printMessage;
                mp.printMessage();
                // instance method reference
                /*HelloMessagePrinter hmp=new HelloMessagePrinter();
                MessagePrinter mp= hmp::printMessage;
                mp.printMessage();*/
        }
}
```

Static Methods in interface.

- static methods are allowed in java8 to be defined in interface. it is due to for just static methods class are so expensive.
- static interface methods would be only called wrt interfaceName.methodName(); wrt implementation class name we would not be able to call.
- static methods are not available to implementation classes by default. hence we need to call with interfacename itself. InterfaceName.staticMethodName();

Streams:- java.util.stream pkg

- Collection :- if we want to represent group of objects as a single entity we use colleciton. or we collect multiple objects of same or differenct types in collection.
- Streams are wrappers for collections and arrays
- Streams are just like pipeline of information. so they don't need storage.
- they can work on existing datasource as collection or array without modifying it.
- The elements of a stream are only visited once during the life of a stream.
- Like an Iterator, a new stream must be generated to revisit the same elements of the source.

Methods of Stream

1) **Filter()** :- this will create a filtered stream of given pridicate.

```
empList.stream().filter(emp->emp.getSalary() >
10000).forEach(System.out::println);
```

2) **map(-)** :- this will return a stream after applying the provided predicate.

```
List<String> vehicles = Arrays.asList("bus", "car", "bicycle", "flight", "train");
vehicles.stream().map(String::toUpperCase).forEach(System.out::println);
```

3) **sorted(-)** :-
   i. this will return stream of sorted collection elements. we need to provide lambda for Comparator or Comparable.
   ii. if elements are not Comparable then we can get ClassCastException.

```
empList.stream()
            .sorted((emp1, emp2)->emp1.getSalary().compareTo(emp2.getSalary()))
            .forEach(System.out::println);
```

4) **limit(number)** :- this will limit the elements while creating stream and create a stream of provided number.

```
Stream.of("bus", "car", "bicycle", "flight",
"train").limit(3).forEach(System.out::println);
```

How to create a stream ?

- Using Stream.of(-)

```
  // create Stream using Stream.of(comma seperated values)
    Stream<Integer> intStream = Stream.of(1,2,3,4,5);
    intStream.forEach(System.out::println);

                  or

    // create stream using array of elements
    Stream<Integer> intStream1 = Stream.of(new Integer[]{1,2,3,4,5});
    intStream1.forEach(System.out::println);
```

- By using stream() method on Collection object.

```
List<String> vehicles = Arrays.asList("bus", "car", "bicycle", "flight", "train");
Stream<String> vStream = vehicles.stream();
```

- by using Stream.generate(-)

```
Stream<UUID> uuidStream = Stream.generate(UUID::randomUUID);
```

➢ if we want to process objects from a collection, we create stream.

```
Stream s=c.stream();

filter(Predicate)

map(Function)

count()

sorted()  // natural sorting order

sorted(Comparator)

min()

max()

forEach(Function)

 toArray() :- this converts stream of objects to array.
```

1) filter returns existing elements filtered.
2) map returns new objects for existing collection.
3) streams are not only applicable for collections,wherever group of values are there, you can go for streams.

Optional class

- Optional is helpful to avoid null values.

```
String input="hellow world..";

                    Optional<String> optional =Optional.ofNullable(input);

                    System.out.println("is present "+optional.isPresent());
                    System.out.println("optional.get():- "+optional.get());   //
throuws NoSuchElementException.
                    System.out.println("optional.orElse"+optional.orElse( "input
value is empty..!"));
                    //optional.orElseGet(s()->"input is empty");

                    //Supplier<String> s=()->"valueNotPresent";
```

Date and Time :- joda time api

- Below are few Date Time api classes.
  we have other classes as
  1. LocalDate:- This will query the system clock in the defaulttime-zone to obtain the current date-time. Below are few static methods
     a. Now()
     b. Of(year,month,dayOfMonth,hour,minute)
  2. LocalTime
  3. LocalDateTime
  4. Period
  5. Year
  6. ZoneId
  7. ZonedDateTime

```
import java.time.LocalDate;
import java.time.LocalTime;

LocalDate ld=LocalDate.now();
System.out.println("local date is :- "+ld);
LocalTime lt=LocalTime.now();
System.out.println("local time is :- "+lt);
```

1) LocalDateTime :- This will query the system clock in the defaulttime-zone to obtain the current date-time.

Example

```
LocalDateTime ldt=LocalDateTime.now();
```

```
        or
LocalDateTime ldt=LocalDateTime.of(2019, Month.JANUARY, 16, 13, 13);

System.out.println(ldt.minusDays(20)); // substracts given long number of days and
returns date.
System.out.println(ldt.plusDays(28));      // adds long number of days and returns
date.
System.out.println(ldt.toString());
System.out.println(ldt.getDayOfMonth());
System.out.println(ldt.getDayOfYear());
System.out.println(ldt.getYear());
System.out.println(ldt.getMonth());
System.out.println(ldt.getHour());
System.out.println(ldt.getMinute());
System.out.println(ldt.getSecond());
System.out.println(ldt.getNano());
System.out.println(ldt.toLocalDate()); //2019-01-16  :- returns LocalDate object
System.out.println(ldt.toLocalTime()); //13:25:02.011 :- returns LocalTime object
```

2) ZonedDateTime :- this will contain zone offset along with date.

```
ZonedDateTime zdt=ZonedDateTime.now();
System.out.println(zdt.toString());  //output :- 2019-01-
16T13:26:41.283+05:30[Asia/Calcutta]


ZoneId zid=ZoneId.systemDefault();
ZonedDateTime.of(LocalDateTime.now(), zid);
System.out.println(zdt.toString());
```

3) LocalDate :- LocalDate currentDate=LocalDate.now(); //2019-01-16

4) LocalTime :- LocalTime currentTime=LocalTime.now(); //15:01:30.820

5) Instant :- introduced Instant class to represent machine readable time formats.

```
Instant stringToInstant= Instant.parse("2019-01-16T14:44:35Z"); // observe the
string format is with 'T' & 'Z' in it.
System.out.println(stringToInstant.toString());
```

6) Period :-

```
Period.of(years, months, days);

                    Period.ofDays(days);

                    Period.ofMonths(months);

                    Period.ofWeeks(weeks);

                    Period.ofYears(years);
                    //to calculate difference between dates.

                    LocalDate currentDate=LocalDate.now();

                    System.out.println(currentDate);


                    LocalDate tenDaysBefore=currentDate.minus(Period.ofDays(10)); // 10
days before date

                    LocalDate twoMonthsAfter=currentDate.plus(Period.ofMonths(2)); //
two months after.


                    Period deffrencBetweenDates=Period.between(tenDaysBefore,
twoMonthsAfter);

                    System.out.println("Years :- "+deffrencBetweenDates.getYears()+"
Months:- "+deffrencBetweenDates.getMonths()+" Days:-
"+deffrencBetweenDates.getDays());
```

7) Duration :-

- As we can calculate difference of dates with help of Period, we can calculate diffrence of time with help of Duration.

```
LocalTime currentTime=LocalTime.now();
LocalTime
after2_HoursAnd30_MinutesAnd15_Seconds=currentTime.plus(Duration.ofHours(2)).pl
us(Duration.ofMinutes(30)).plusSeconds(15);
System.out.println(after2_HoursAnd30_MinutesAnd15_Seconds.toString());
//17:47:47.367
```

Example: -calculating difference between execution in seconds and miliseconds.

```
                    Long beforeStartingExecution=Instant.now().getEpochSecond();
                    //calling business logic method.
                    someBusinessLogic();
                    Long afterExecutionFinished=Instant.now().getEpochSecond();
                    Long difference= afterExecutionFinished-beforeStartingExecution;

                    System.out.println("executionTime in seconds :- "+difference);

                    Long beforeStart=Instant.now().toEpochMilli();
                    someBusinessLogic();
                    Long afterComplete=Instant.now().toEpochMilli();
                    Long diff=afterComplete-beforeStart;
                    System.out.println("executionTime in miliSeconds :- "+diff);

                    public static void someBusinessLogic(){

                    try {
                              TimeUnit.SECONDS.sleep(3);
```

```
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
```

Example : Converting string to date ?

```
// means string date is in "yyyy-mm-dd" format

        String dateStr = "2015-11-18";

        LocalDate dtObj = LocalDate.parse(dateStr);

        System.out.println("Date object: "+dtObj);


        // convert it using formatter

        DateTimeFormatter dateFormatter =
DateTimeFormatter.ofPattern("dd/MM/yyyy");

        LocalDate fdtObj = LocalDate.parse("18/11/2015", dateFormatter);

        System.out.println("Date object: "+fdtObj);
```

Predicate: interface.

- predicates are part of java.util.function.
- we can create lambda expressions for condition checking logic which returns boolean to make consise code.
- by default, return type of predicate is always boolean.

```
        Predicate<String> p=s->s.length()>5;
        Predicate<Integer> p=i->i%2==0;
        Predicate<Employee> result=e->e.getSalary()>10000 &&
        e.getLocation().equals("pune");
```

```
e.g.

import java.util.function.Predicate;

public class PredicateTest {


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

```
        // TODO Auto-generated method stub
        String[] nameArray={"Nag","Chiranjeevi","Venkatesh",
"Balaiah","sunny","Katrina"};
        Predicate<String> p=s->s.length()>5;

            for(String s:nameArray)
           {
                if(p.test(s))
                System.out.println(s);
            }
         }
        }
```

- 'and','or' are the predicate joining methods.
- negate is to get negatinve result of a predicate.

Function: interface.

- if result required is boolean then we use predicate interface.
- if result is other than boolean then we create Function interface object and provide input type in generic symbol.

Consumer: interface.

Supplier: interface.

BiPredicate,BiFunction,BiConsumer

Primitive Predicate Types.

- IntPredicate,DoublePredicate,LongPredicate.

Primitive Function Types.

- IntFunction, DoubleFunction,LongFunction IntToDoubleFunction,IntToLongFunction etc...

Primitive Consumer Types.

- IntConsumer,DoubleConsumer,LongConsumer
    - acceptAsInt()

Primitive Supplier Types.

- BooleanSupplier,IntSupplier,LongSupplier,DoubleSupplier
    - getAsDouble()

- IntBinaryOperator,LongBinaryOperator,DoubleBinaryOperator
- IntUnaryOperator