Java8 feature

Functional programming

JAVA8 Features:

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

why java1.8 ?

1)to simplify the programming....more concise.

2)to utilizes the functional programming in java

3)to enable parallel programming..etc

readable,maintainable,concise code.

to use api's very easily and effectively.

public int add(int a,int b)

{

return a+b;

}

(a,b)-> a+b;

no method name,parameters,return ,

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

1)Lambda Expressions-->

2)Functional Interface,interface one abstract method...

3)Default Methods and static methods

4)Predefined Functional Interfaces/runnable

Predicate

Function

Consumer

Supplier

etc..

5)Double colon Operator(::);

Method Reference

Constructor Reference

6.streams(Terminal and Intermediate Operations)stream....

7.Date and Time APi

8.Optional class

1)Lambda Expressions:

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

1930

LISP

-------

Python

Lisp

C

C++

RUBY

Scala

Finally in java also

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

The main Objective of Lambda EXPression:

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

to bring benefits of functional programming

into java.

What is Lambda Expression:

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

pubilc void m1(int a,int b){

return a+b;

}

(a,b)->a+b;

It is an anonymous(Nameless) function.

Nameless

Without return type

without modifiers

Ex1:

====

public void m1(){

System.out.println("Welcome");

}

into lambdaExpression

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

or

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

note:

=====

if Lambda expression contains one line curely braces are optional

eg2:

====

public void m1(int a,int b){

System.out.println(a+b);

}

convert into lambda Expression

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

eg3:

====

A

public int squareIt(int n){

return n\*n;

}

convert into Lambda Expression.

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

(n) -> n\*n;

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

note:

=====

1)if you want to return the value using

return keyword the braces are mandatory

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

eg4:

====

public int m1(String s){

return s.length();

}

convert into lambda expression

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

s->s.length();

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

FI --Functional Interface to call lambda Expression

word @FunctionalInterface->default +static+SAM

old

Runnable ==> run()

Comparable ==> compareTo()-->NSO

Comparator ==>compare()         -->CSO

Callable==>call()---->

note:

=====

default method +static method

@FunctionalInterface 1 abstract+static \_

An interface which containse only one Abstract Method

is called Functional Interface.

Once we write Lambda Expression to invoke

we need Functional Interfaces

can i take default methods &static methods

in Functional Interface?

yes but Exact only abstract method and

any no of default and static methods

=

interface A

{

default void m1(){}

}

class Test{

static String name="suresh";

}

Test.name.length();

class System{

static PrintStream out=

}

Example on Stream

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

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.Stream;

public class StreamEx {

public static void main(String[] args) {

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

al.add(10);

al.add(40);

al.add(30);

al.add(15);

al.add(5);

System.out.println("gg"+al);//

List<Integer> l = al.stream().filter(i->i%2==0).collect(Collectors.toList());

System.out.println("filter to print double value "+l);

List<Integer> l1 = al.stream().map(i->i\*2).collect(Collectors.toList());

System.out.println("increment the existing value using map "+l1);

long cn =al.stream().count();

System.out.println("it will count no of elements "+cn);

List<Integer> l2 = al.stream().sorted((i1,i2)->(i1<i2)?1:(i1>i2)?-1:0).collect(Collectors.toList());

System.out.println("displayed in descending order "+l2);

Integer minVal = al.stream().min((i1,i2)->-i1.compareTo(i2)).get();

System.out.println("minimumvalue is ... "+minVal);

Integer maxVal = al.stream().max((i1,i2)->-i1.compareTo(i2)).get();

System.out.println("Maximum..value is ... "+maxVal);

//printing using foreach

al.stream().forEach(i->{

System.out.println("the elements are "+i);

});

//to array list to array

Object []a=al.stream().toArray();

System.out.println("the array elements are ");

for(Object a1:a) {

System.out.println(a1);

}

System.out.println("the array elements are using forEach method");

Stream.of(a).forEach(i->{

System.out.println(i);

});

}

}

java.util.function

Predefined Functional Interfaces:40+

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

Predicate SAM one abstract method test()

Consumer

Supplier

Function

Predicate(i)

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

Where ever boolean condition checking is required we can use this predicate FI

input can be anything but result of predicate test method is boolean.

interface Predicate<T>{

public boolean test(T t);

}

To check weather given number is even or not

public boolean test(int i)

{

if(i%2==0)

{

return true;

}

else{

return false;

}

}

lamda:

(Integer I)->I%2

I->I%2;-->

Predicate<Integer> predicate=i->i%2==0

syso(predicate.test(10))

Ex:

====

p1---> will check weather given number is even r not

p2--->will check weather given number is greater than 10 r not

p1.and(p2).test(34)-->true(predicate joining)

Function:(FI)

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

it will do some operation and returns result(any type)

input--->operation--->result

4--->cube it-->64

interface Function<T,R>{

public R apply(T t);

}

ex:-

----

Function<Integer, Integer> f=i->i\*i;

System.out.println(f.apply(20));

ex:-

-----

String s="welcome";

Function<String, Integer> f=i->i.length();

System.out.println(f.apply(s));

ex:-

-----

String s="welcome";

Function<String, String> f=i->i.toUpperCase();

System.out.println(f.apply(s));

ex:

-------

Function<Integer, Integer> f=i->2\*i;

Function<Integer, Integer> f1=i->i\*i\*i;

System.out.println(f.andThen(f2).apply(2));

System.out.println(f.compose(f2).apply(2));//function chaining

Consumer:

-----------

Consumer <T>--->void

interface Consumer<Integer i>{

public void accept(T t);

}

Supplier:

=========

\*it will not take any input...

interface Supplier{

public R get();

}

IntPredicate

Predicate-->test()-->boolean

input can be anything but result of predicate test method is boolean.

Function --->apply()--->anything

input can be anything and result of Function apply() method is also anyting.

Consumer --->accept()-->void

input can be anything but result of Consumer accept() method is nothing.

supplier--->get()-->anything

\*it will not take any input can return anything.

\*\*checking given number is even or odd we can use predicate

\*\*check additon of two numbers is even or odd ,but predicate test() can take only one parameter

so we need to go for BiPredicate FI

interface BiPredicate<T1 t1,T2 t2>{

public boolean test(T t);

}

Ex:

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

syso(p.test(12,11));

Primitive version of Finctional Interface:

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

Primitive Predicate Types:

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

IntPredicate

DoublePredicate

LongPredicate

Primitive Function Types:

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

DoubleFunction-->double as input and return can be anything

IntFunction--->Int as input and return can be anything

ToIntFunction--->input:anything,return:int-->applyAsInt()

LongFunction--->long as input and return can be anything

DoubleToIntFunction-->input:double,return:int-->applyAsInt()

DoubleToLongFunction-->input:double,return:long-->applyAsLong()

IntToDoubleFuntion--->input:int,return:double-->applyAsDouble()

IntToLongFunction--->input:int,return:Long-->applyAsLong()

LongToIntFunction--->input:Long,return:int-->applyAsInt()

LongToDoubleFunction--->input:int,return:Long-->applyAsDouble()

ToIntFunction--->input:anything,return:int-->applyAsInt()

ToLongFunction--->input:anything,return:Long-->applyAsLong()

ToDoubleFunction--->input:anything,return:Double-->applyAsDouble()

ToIntBiFunction--->input(2):anything,return:int-->applyAsInt()

ToLongBiFunction--->input(2):anything,return:Long-->applyAsLong()

ToDoubleBiFunction--->input(2):anything,return:Double-->applyAsDouble()

Primitive types of Consumer:

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

DoubleConsumer--->input:double,return:void

IntConsumer--->input:int,return:void

LongConsumer--->input:long,return:void

primitive Types of supplier:

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

BooleanSupplier

IntSupplier

LongSupplier

DoubleSupplier

......etc

Method reference and constructor reference:

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

::-->code reusability

alternative for lamda expression

classname::myMethod--->Method reference static

classname::myMethod--->Method reference static

classname::new-->constructor reference

Date and Time API:

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

until java 7

Date,Calender,TimeZone(performance issues)

JODA API(joda.org)

java.time package

stream api project

ForDate:

---------

LocalDate date=LocalDate.now();

sop(date)

getDayOfMonth()getMonthValue()getYear()....

For Time:

-----------

LocalTime time=LocalTime.now();

sop(time)

getHour(),getMinute(),getSecond(),getNano()....

For Date/Time:

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

LocalDateTime dateTime=LocalDateTime.now();

sop(dateTime)

dob

----

LocalDateTime dateTime=LocalDateTime.of(1993,08,10,10,45);

sop(dateTime)

plusMonths()

minusMonths()...........

Period Class:

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

LocalDate date=LocalDate.now();

LocalDate bDate=LocalDate.of(1993,11,10);

Period p=Period.between(bDate,date);

sopf("ur age %d years,%d months,%d days",p.getYears(),p.getMonths(),p.getDays());

Year lyear=Year.of(Integer.parseInt(args[0]));

if(lyear.isLeep())...

Year class has isLeep() method check leap year

For timezone:

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

ZoneId zone=ZoneId.systemDefault();

sop(zone)//Asia/Calcutta

ZoneId zone=ZoneId.of("America/New\_York");

ZonedDateTime zt=ZonedDateTime.now(zone);

System.out.println(zt);//Asia/Calcutta

IMP:

=====

LocalDate bDate=LocalDate.of(yyyy,mm,dd);

String to LocaDate-->parse method we have to use

LocaDate to String-->format method we have to use

LocalDate.parse

Case-1

LocalDate ld=LocalDate.of(2019,8,10);

ld.plusDays(10);

System.out.println(ld);

case-2

LocalDate ld=LocalDate.parse("2019-08-10");

String s=ld.format(DateTimeFormatter.ISO\_LOCAL\_DATE\_TIME);

System.out.println(s);

case-3

LocalDateTime dt=LocalDateTime.of(2019,8,10,12,45);

dt.plusDays(25);

dt.plusMonths(2);

System.out.println(dt.format(DateTimeFormatter.ISO\_DATE\_TIME));

case-4

sLocalDate ld=LocalDate.now();

LocalDate ld1=LocalDate.of(2019,8,10);

LocalDate ld2=LocalDate.parse("2019-08-10",DateTimeFormatter.ISO\_DATE);

System.out.println(ld);

System.out.println(ld1);

System.out.println(ld2);

Note:LocalDate,LocalTime,LocalDateTime are immutable Objects