**Lambda Expression**

* To bring benefits of functional programming into java.
* It is an anonymous function. (Nameless)
* This should be:
  + Without name
  + Without return type
  + Without modifiers

Example to write Lambda Expression:-

1. public void m1()

{

s.o.p(“Hello”);

}

L.E. : () 🡪 { s.o.p(“Hello”);}

1. public void m1(int a, int b)

{

s.o.p(a+b);

}

We can remove curly braces when single line code is there

L.E. : (int a, int b) 🡪 s.o.p(a+b); argument type is not mandatory

Or (a,b) 🡪 s.o.p(a+b);

1. public int squareIt(int n)

{

return n\*n;

}

Parenthesis is optional when only one argument is there

return keyword must required when curly braces is present or else optional

L.E.: (int n) 🡪 { return n\*n ; }

Or (int n) 🡪 n\*n;

Or (n) 🡪 n\*n;

Or n 🡪 n\*n;

1. public void m1(String s)

{

return s.length();

}

L.E. : s 🡪 s.length();

Rules for Lambda function:

* Without curly braces no need to use return keyword, complier will consider returned value automatically.
* Within curly braces if we want to return some value compulsory we should use return statement.
* Parenthesis is optional when only one argument is present.

**Functional Interface:**

An interface having SAM (Single Abstract Method) is known as Functional interface.

e.g.:-

Runnable 🡺 run()

Comparable 🡺 compareTo()

Comparator 🡺 compare()

ActionListener 🡺 actionPerformed()

Callable 🡺 call()

We can use annotation for explicit declaration.

**e.g.**

@FunctionalInterface

interface Interf

{

-----------

}

Below are few examples of valid/invalid Functional interfaces:

1. interface A

{

Public void m1(); // abstract method

default void m2()

{

}

public static void m3()

{

-----------

}

}

This is a valid Functional interface.

1. @FunctionalInterface

interface A

{

Public void m1();

}

@FunctionalInterface

interface B extends A

{

}

This is a valid one.

1. @FunctionalInterface

interface A

{

Public void m1();

}

@FunctionalInterface

interface B extends A

{

Public void m1();

}

This one is valid.

1. @FunctionalInterface

interface A

{

Public void m1();

}

@FunctionalInterface

interface B extends A

{

Public void m2();

}

This one is not valid.

1. @FunctionalInterface

interface A

{

Public void m1();

}

interface B extends A

{

Public void m2();

}

This one is valid.

**Lambda Expression with Functional Interfaces:**

Interface without Lambda expression:

**public** **class** MyTest {

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

Interf i = **new** Demo();

i.m1();

}

}

**interface** Interf

{

**public** **void** m1();

}

**class** Demo **implements** Interf {

**public** **void** m1() {

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

}

}

Output - Hello

Interface with Lambda expression:

**public** **class** MyTest {

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

Interf i = () -> System.***out***.println("Lambda");

i.m1();

}

}

**interface** Interf

{

**public** **void** m1();

}

Output – Lambda

**public** **class** MyTest {

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

Interf i = x ->System.***out***.println("The result is: " + 2\*x);

i.add(10);

}

}

**interface** Interf

{

**public** **void** add(**int** a);

}

Output: The result is: 20

**public** **class** MyTest {

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

Interf i = (x,y) ->System.***out***.println("The result is: " + (y+x));

i.add(10,20);

}

}

**interface** Interf

{

**public** **void** add(**int** a, **int** b);

}

Output : The result is: 30

* Lambda expressions are always associated with Functional interface.
* If Functional interface is not present then we cannot use Lambda expression.
* At the time of compilation lambda expression will be converted to private method. It will never generate separate .class file.

**Threading**

By implementing Runnable interface without Lambda expression:

**public** **class** MyTest {

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

MyRunnable r = **new** MyRunnable();

Thread t = **new** Thread(r);

t.start();

**for**(**int** i=0; i<5; i++) {

System.***out***.println("Main thread");

}

}

}

**class** MyRunnable **implements** Runnable{

**public** **void** run() {

**for** (**int** i=0; i<5; i++) {

System.***out***.println("Child thread");

}

}

}

Here we will get mixed output due to multiple threads.

Output: Main thread

Child thread

Child thread

Child thread

Child thread

Child thread

Main thread

Main thread

Main thread

Main thread

Using Lambda expression for Threading:

**public** **class** MyTest {

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

Runnable r = () -> {

**for**(**int** i=0; i<5; i++) {

System.***out***.println("Child Thread");

}

};

Thread t = **new** Thread(r);

t.start();

**for**(**int** i=0; i<5; i++) {

System.***out***.println("Main Thread");

}

}

}

**Lambda Expression with Collection**

Comparator interface having method called compare().

int compare(Object obj1, Object obj2)

this method returns :

* negative value if obj1 has to come before obj2
* positive value if obj1 has to come after obj2
* 0 if obj1 and obj2 are equal

Program for sorting without using Lambda expression:

**public** **class** MyTest {

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

ArrayList<Integer> l = **new** ArrayList<Integer>();

l.add(10);

l.add(5);

l.add(25);

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

Collections.*sort*(l, **new** MyComparator());

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

}

}

**class** MyComparator **implements** Comparator<Integer>{

**public** **int** compare(Integer I1, Integer I2) {

**if**(I1<I2) {

**return** -1;

}

**else** **if**(I1>I2) {

**return** 1;

}

**else** {

**return** 0;

}

}

}

Output –

[10, 5, 25]

[5, 10, 25]

With Lambda Expression:

**public** **class** MyTest {

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

ArrayList<Integer> l = **new** ArrayList<Integer>();

l.add(10);

l.add(5);

l.add(25);

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

Comparator<Integer> c = (I1,I2) -> (I1<I2)?-1: (I1>I2)?1: 0;

Collections.*sort*(l,c);

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

}

}

Output –

[10, 5, 25]

[5, 10, 25]

I1, I2 are the parameters to compare method of Integer type.

? symbol is used for return statement.

**Streams**

* If we want to process objects from the collection then we require Streams.
* If we want to get the stream then stream() itself the method we have to use.

e.g. Stream s = c.stream();

**Program 1**: Get even numbers from an ArrayList

**public** **class** MyTest {

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

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

al.add(10);

al.add(5);

al.add(25);

al.add(12);

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

List<Integer> l2 = al.stream().

filter(i->i%2==0).collect(Collectors.*toList*());

System.***out***.println(l2);

}

}

Output –

[10, 5, 25, 12]

[10, 12]

**Program 2**: For every Student add 5 marks as grace

**public** **class** MyTest {

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

ArrayList<Integer> marks = **new** ArrayList<Integer>();

marks.add(10);

marks.add(50);

marks.add(25);

marks.add(85);

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

List<Integer> l2 = marks.stream().

map(i->i+5).collect(Collectors.*toList*());

System.***out***.println(l2);

}

}

Output –

[10, 50, 25, 85]

[15, 55, 30, 90]

filter() – When we want to apply some condition or want to put some filter on the objects.

map() – When we want to perform operation on the objects.

**Different methods of Stream:**

* stream()
* filter()
* map()
* collect()
* count()
* sorted() -> Default natural sorting order
* sorted(comparator) -> Customized sorting order
* min(comparator)
* max(comparator)
* forEach()
* toArray()

**Program 3**: How many failed students are there in a Arraylist?

**public** **class** MyTest {

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

ArrayList<Integer> marks = **new** ArrayList<Integer>();

marks.add(70);

marks.add(50);

marks.add(25);

marks.add(15);

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

**long** noOfFailedStudents = marks.stream().filter(m->m<35).count();

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

}

}

Output –

[70, 50, 25, 15]

2

**Program 4**: Sort all the marks on Arraylist?

**public** **class** MyTest {

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

ArrayList<Integer> marks = **new** ArrayList<Integer>();

marks.add(10);

marks.add(50);

marks.add(25);

marks.add(45);

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

List<Integer> sortedList =

marks.stream().sorted().collect(Collectors.*toList*());

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

}

}

Output –

[10, 50, 25, 45]

[10, 25, 45, 50]

Comparator interface having method -> compare(obj1, obj2)

int compare(obj1, obj2):

this method returns :

* negative value if obj1 has to come before obj2
* positive value if obj1 has to come after obj2
* 0 if obj1 and obj2 are equal

Logic for descending order:

(i1,i2) -> (i1<i2)?1 : (i1>i2)?-1 : 0

**Program 5**: Sorting of Arraylist in descending order?

**public** **class** MyTest {

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

ArrayList<Integer> marks = **new** ArrayList<Integer>();

marks.add(10);

marks.add(50);

marks.add(25);

marks.add(45);

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

List<Integer> sortedList = marks.stream().

sorted((i1,i2)->(i1<i2)?1 : (i1>i2)?-1 : 0).collect(Collectors.*toList*());

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

}

}

Output –

[10, 50, 25, 45]

[50, 45, 25, 10]

**Default Natural Soring Order:**

* + This is implemented by Comparable interface.
  + Comparable interface contains compareTo() method.
  + If we are not passing anything inside sorted() then inside Comparable, compareTo() method will be called.
  + When we are not passing parameter for sorted() then internally below operations happens.

(i1,i2) 🡪 i1.compareTo(i2)

* + So using this we can perform sorting.

Ascending order:

List<Integer> sortedList = marks.stream().

sorted((i1,i2) 🡪 i1.compareTo(i2)).collect(Collectors.*toList*());

Descending order:

List<Integer> sortedList = marks.stream().

sorted((i1,i2) 🡪 i2.compareTo(i1)).collect(Collectors.*toList*());

**OR** by using - symbol

List<Integer> sortedList = marks.stream().

sorted((i1,i2) 🡪 - i1.compareTo(i2)).collect(Collectors.*toList*());

**Program 6**: Sorting of sting objects? (Default natural sorting)

**public** **class** MyTest {

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

ArrayList<String> al = **new** ArrayList<String>();

al.add("Lipu");

al.add("Bond");

al.add("Deepak");

al.add("Kshriod");

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

List<String> sortedList =

al.stream().sorted().collect(Collectors.*toList*());

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

}

}

Output –

[Lipu, Bond, Deepak, Kshriod]

[Bond, Deepak, Kshriod, Lipu]

**Program 7**: Sorting of sting objects? (Reverse of natural sorting order)

**public** **class** MyTest {

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

ArrayList<String> al = **new** ArrayList<String>();

al.add("Lipu");

al.add("Bond");

al.add("Deepak");

al.add("Kshriod");

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

List<String> sortedList = al.stream().

sorted((s1,s2)-> s2.compareTo(s1)).collect(Collectors.*toList*());

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

}

}

Output –

[Lipu, Bond, Deepak, Kshriod]

[Lipu, Kshriod, Deepak, Bond]

Reverse of natural sorting order

(s1,s2)-> s2.compareTo(s1)

Or by using - symbol

(s1,s2)-> - s1.compareTo(s2)

**Program 8**: Arrange the strings in terms of their length (Increasing order).

**public** **class** MyTest {

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

ArrayList<String> al = **new** ArrayList<String>();

al.add("A");

al.add("AAAAAAAAAAAA");

al.add("AAAAA");

al.add("AAA");

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

Comparator<String> c = (s1, s2) -> {

**int** l1 = s1.length();

**int** l2 = s2.length();

**if** (l1<l2) **return** -1;

**else** **if** (l1>l2) **return** +1;

**else** **return** s1.compareTo(s2);

};

List<String> sortedList =

al.stream().sorted(c).collect(Collectors.*toList*());

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

}

}

Output –

[A, AAAAAAAAAAAA, AAAAA, AAA]

[A, AAA, AAAAA, AAAAAAAAAAAA]

**Program 9**: To get minimum & maximum number from an ArrayList by Ascending order.

**public** **class** Test {

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

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

al.add(10);

al.add(5);

al.add(25);

al.add(85);

al.add(30);

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

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

System.***out***.println("minimum: " +min);

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

System.***out***.println("maximum: "+max);

}

}

Output –

[10, 5, 25, 85, 30]

minimum: 5

maximum: 85

**Program 10**: To get minimum & maximum number from an ArrayList by Descending order.

**public** **class** Test {

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

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

al.add(10);

al.add(5);

al.add(25);

al.add(85);

al.add(30);

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

Integer min = al.stream().

min((i1,i2) -> - i1.compareTo(i2)).get();

System.***out***.println("minimum: " +min);

Integer max = al.stream().

max((i1,i2) -> - i1.compareTo(i2)).get();

System.***out***.println("maximum: "+max);

}

}

Output –

minimum: 85

maximum: 5

**forEach() loop:**

**public** **class** Test {

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

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

al.add(10);

al.add(5);

al.add(25);

al.add(85);

al.add(30);

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

al.stream().forEach(System.***out***::println);

}

}

Output –

[10, 5, 25, 85, 30]

10

5

25

85

30

**Program 11**: Print the square value of each element of ArrayList

**public** **class** Test {

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

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

al.add(10);

al.add(5);

al.add(25);

al.add(85);

al.add(30);

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

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

System.***out***.println("The square of " + i + " is: " + (i\*i));

});

}

}

Output –

The square of 5 is: 25

The square of 25 is: 625

The square of 85 is: 7225

The square of 30 is: 900

**toArray() :**

* This method is used to convert stream of objects into array.

**Program 12**: Convert an ArrayList to Integer array?

**public** **class** Test {

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

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

al.add(10);

al.add(5);

al.add(25);

al.add(85);

al.add(30);

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

Integer[] i = al.stream().toArray(Integer[]::**new**);

**for**(Integer a:i) {

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

}

}

}

Output –

[10, 5, 25, 85, 30]

10

5

25

85

30

**Q.** How to get stream of Array?

Ans. By using Stream.of()

**Stream.of() :**

e.g.

**public** **class** Test {

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

Integer[] i = {10,56,24,72,64,28};

Stream.*of*(i).forEach(System.***out***::println);

}

}

Output –

10

56

24

72

64

28