**Anonymous inner class vs Lambda Expression**

Anonymous inner class can extend a normal class.

Anonymous inner class can extend an abstract class.

Anonymous inner 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).

AIC is more powerful than LE

**Default Method/virtual extension method/defender mthod**

Until 1.7v ,every method inside interface is always public and abstract.

1.8v:- default methods + static methods

1.9v: private method

Defender means it gonna provide protection to all impl classes.

**package** test;

**interface** InterF

{

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

**public** **void** m2();

**public void** m3();/CTE

}

**class** demo **implements** InterF

{

**public** **void** m1() {};

**public** **void** m2() {};

}

**class** demo1 **implements** InterF

{

**public** **void** m1() {};

**public** **void** m2() {};

}

Solution is :

**package** test;

**interface** InterF

{

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

**public** **void** m2();

**default** **void** m3()

{

System.***out***.println("default method called");

}

}

**class** demo **implements** InterF

{

**public** **void** m1() {};

**public** **void** m2() {};

}

**class** demo1 **implements** InterF

{

**public** **void** m1() {};

**public** **void** m2() {};

}

Default means this method having default implementation .it not a modifier.

Without affecting implementation classes if we want to add extra functionalities then it is required.

Default is not modifier here since without mentioning anything complier consider this as default.

**package** test;

**interface** InterF

{

**default** **void** m1()

{

System.***out***.println("default method");

}

}

**public** **class** demo **implements** InterF

{

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

{

System.***out***.println("overriding version of default method");

}

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

{

demo d=**new** demo();

d.m1();

}

}

While overriding we have make it public from default. Default is only allowed inside interface not in class.

**package** test;

**interface** InterF

{

**default** **void** m1()

{

System.***out***.println("default method");

}

**default** **int** hashCode() ////CTE {

**return** 5;

}

}

**public** **class** demo **implements** InterF

{

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

{

}

}

We are not allowed to declare all object class method as default method. It’s already present on demo class.

Multiple inheritance is possible only through default method like below.

**package** test;

**interface** left

{

**default** **void** m1()

{

System.***out***.println("left m1 default method");

}

}

**interface** right

{

**default** **void** m1()

{

System.***out***.println("right m1 default method");

}

}

**public** **class** demo **implements** left,right

{

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

{

System.***out***.println("our own method");

left.**super**.m1();

right.**super**.m1();

}

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

{

demo d=**new** demo();

d.m1();

}

o/p

our own method

left m1 default method

right m1 default method

**Static Methods**

To define general utility method interface statis method came.

For static methods, no need to create a object. So from 1.8v onwards we can use static methods from interface since interface is less costly than class(contain constructor , objects etc.) and takes less memory than class.

**package** test;

**interface** InterF

{

**public** **static** **void** m1()

{

System.***out***.println("interface static method");

}

}

**class** demo **implements** InterF

{

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

{

InterF.*m1*();//valid

m1();//invalid

test.m1();//invalid

demo d=**new** demo();

d.m1();//invalid

}

}

Interface should be called by interface name only. By Implementation class cannot call.

**package** test;

**interface** InterF

{

**public** **static** **void** m1()

{

System.***out***.println("interface static method");

}

}

**class** demo

{

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

{

InterF.*m1*();

// m1();

// test.m1();

// demo d=new demo();

// d.m1();

}

}

o/p:- interface static method

if a class is not implementing also we can call s static method of interface.

We can declare main method inside interface also.

**package** test;

**interface** InterF

{

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

{

System.***out***.println("interface main method");

}

}

To define general utility method(add,sub,product,div etc) we should go for interface instead of class.

**interface** InterF

{

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

{

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

}

**public** **static** **void** sub(**int** a,**int** b)

{

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

}

**public** **static** **void** prodcut(**int** a,**int** b)

{

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

}

**public** **static** **void** div(**int** a,**int** b)

{

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

}

}

If all are static and not related to object then why should we go for class since more memory will be consumed.

**Predefined FI:**

To make lambda expression as common coding activity there are some predefined FI since LE can be called by FI only.

Eg:

Predicate

Function

Consumer

Supplier

Two argument predefined FI

BiPredicate

BiFunction

BiConsumer

Primitive FI

IntPredicate

IntFunction

IntConsumer

All predefined FIs are available on java.util.Function package.

Predicate(I):

interface Predicate<T>

{

public boolean test(T t);

}

Whenever there is a conditional check then we should go for Predicate

**public** **boolean** test(Integer I)

**if**(I%2==0)

{

**return** **true**;

}

**else**

{

**return** **false**;

}

LE:

Integer I)->I%2==0

I->I%2==0

**import** java.util.function.\*;

**class** demo

{

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

{

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

System.***out***.println(p1.test(40));

System.***out***.println(p1.test(45));

}

}

o/p:

true

false

we have if else but we should go for predicate to do check more condition by LE. We can’t write 10 if else condition but in one line 10 condition can be checked on predicate by LE.

Eg:-

Predicate<Employee> em=e->e.salary>10000 && e.isHavingGF==**true** && cond-3;

System.***out***.println(em.test(e));

**Write a predicate to check whether length of string is greater than 5 or not.**

Predicate<String> p=s->s.length()>5;

**import** java.util.function.\*;

**public** **class** demo

{

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

{

String[] s= {"Nag","dev","saswat","rakesh"};

Predicate<String> p =s1->s1.length()>5;

**for** (String s1: s)

{

**if** (p.test(s1))

{

System.***out***.println(s1);

}

}

}

}

o/p

saswat

rakesh

e.g

**import** java.util.ArrayList;

**import** java.util.function.\*;

**class** Employee

{

String name;

**double** salary;

**public** Employee(String name, **double** salary)

{

**this**.name = name;

**this**.salary = salary;

}

}

**public** **class** demo

{

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

{

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

l.add(**new** Employee("saswat",1000));

l.add(**new** Employee("rakesh",2000));

l.add(**new** Employee("ramesh",3000));

l.add(**new** Employee("saswat2",4000));

l.add(**new** Employee("saswat3",5000));

Predicate<Employee> p=e->e.salary>3000;

**for** (Employee e1:l)

{

**if**(p.test(e1))

{

System.***out***.println(e1.name+":"+e1.salary);

}

}

}

}

Combine two predicate into single predicate

P1: check whether number is even or odd

P2: check whether number is > 10

P1.and(p2).test(34)

P1.or(p2).test(34)

P1.negate.test(34)/ opposite of p1

**import** java.util.ArrayList;

**import** java.util.function.\*;

**public** **class** demo

{

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

{

**int**[] x= {0,5,10,40,60};

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

Predicate<Integer> p2=i->i>10;

System.***out***.println("number which are even and >10: ");

**for**(**int** x1:x)

{

**if**(p1.and(p2).test(x1))

{

System.***out***.println(x1);

}

}

}

}

**import** java.util.ArrayList;

**import** java.util.function.\*;

**public** **class** demo

{

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

{

**int**[] x= {0,5,10,40,60};

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

Predicate<Integer> p2=i->i>10;

System.***out***.println("number which are not even: ");

**for**(**int** x1:x)

{

**if**(p1.negate().test(x1))

{

System.***out***.println(x1);

}

}

}

}

If there is some input ,we should perform some operation and produce result as non-boolean type then we should go for Function instead of Predicate.

**Function(I)**

Interface Function<T,R>

{

Public R apply(T t)

}

**package** test;

**import** java.util.ArrayList;

**import** java.util.function.\*;

**public** **class** demo

{

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

{

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

System.***out***.println(f.apply(4));

}

}

o/p= 16

**import** java.util.ArrayList;

**import** java.util.function.\*;

**public** **class** demo

{

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

{

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

System.***out***.println(f.apply("saswat"));

}

}

o/p: 6

**package** test;

**import** java.util.ArrayList;

**import** java.util.function.\*;

**public** **class** demo

{

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

{

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

System.***out***.println(f.apply("saswat"));

}

}

o/p: SASWAT

**import** java.util.ArrayList;

**import** java.util.function.\*;

**class** Student

{

String name;

**int** marks;

**public** Student(String name, **int** marks)

{

**this**.name = name;

**this**.marks = marks;

}

}

**public** **class** demo

{

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

{

Student[] sf= {**new** Student("saswat",100),**new** Student("rakesh",60)};

Function<Student,String> f=s->{**int** marks=s.marks;

String grade="";

**if** (marks>=80) grade="A";

**else** **if**(marks>=60) grade="B";

**else** **if**(marks>=50) grade="C";

**else** grade="F";

**return** grade;};

**for** (Student s1:sf)

{

System.***out***.println("student name:"+s1.name);

System.***out***.println("student marks:"+s1.marks);

System.***out***.println("grade :"+f.apply(s1));

}

}

}

o/p

student name:saswat

student marks:100

grade :A

student name:rakesh

student marks:60

grade :B

**import** java.util.ArrayList;

**import** java.util.function.\*;

**class** Student

{

String name;

**int** marks;

**public** Student(String name, **int** marks)

{

**this**.name = name;

**this**.marks = marks;

}

}

**public** **class** demo

{

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

{

Student[] sf= {**new** Student("saswat",100),**new** Student("rakesh",60)};

Function<Student,String> f=s->{**int** marks=s.marks;

String grade="";

**if** (marks>=80) grade="A";

**else** **if**(marks>=60) grade="B";

**else** **if**(marks>=50) grade="C";

**else** grade="F";

**return** grade;};

Predicate<Student> p=s->s.marks>=90;

**for** (Student s1:sf)

{

**if**(p.test(s1))

{

System.***out***.println("student name:"+s1.name);

System.***out***.println("student marks:"+s1.marks);

System.***out***.println("grade :"+f.apply(s1));

}

}

}

}

o/p

student name:saswat

student marks:100

grade :A

**function chaining:**

F1.andThen(F2).apply(i);first f1 then f2

F1.compose(F2).apply(i);//first f2 then f1

For function chaining we should go for Consumer.

**package** test;

**import** java.util.ArrayList;

**import** java.util.function.\*;

**public** **class** demo

{

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

{

Function<Integer,Integer> f1=i->i\*2;

Function<Integer,Integer> f2=i->i\*i;

System.***out***.println(f1.andThen(f2).apply(4));

System.***out***.println(f1.compose(f2).apply(4));

}

}

Op

64

32

**Consumer:**

Consumer<T>--🡪 consume i.e return void

interface Consume<T>

{

Public void accept(T t);

}

**import** java.util.function.\*;

**public** **class** demo

{

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

{

Consumer<String> c=s->System.***out***.println(s);

c.accept("saswat");

}

}

It’s gonna consume i.e just print it .no need to do anything.

Op:

saswat

**import** java.util.ArrayList;

**import** java.util.function.\*;

**class** Student

{

String name;

**int** marks;

**public** Student(String name, **int** marks)

{

**this**.name = name;

**this**.marks = marks;

}

}

**public** **class** demo

{

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

{

Student[] sf= {**new** Student("saswat",100),**new** Student("rakesh",60)};

Function<Student,String> f=s->{**int** marks=s.marks;

String grade="";

**if** (marks>=80) grade="A";

**else** **if**(marks>=60) grade="B";

**else** **if**(marks>=50) grade="C";

**else** grade="F";

**return** grade;};

Predicate<Student> p=s->s.marks>=90;

Consumer<Student> c=s1->

{

System.***out***.println("student name:"+s1.name);

System.***out***.println("student marks:"+s1.marks);

System.***out***.println("grade :"+f.apply(s1));

};

**for** (Student s1:sf)

{

**if**(p.test(s1))

{

c.accept(s1);

}

}

}

}

Op:

student name:saswat

student marks:100

grade :A

**consumer chaining**

**import** java.util.function.\*;

**class** Movie

{

String name;

**public** Movie(String name)

{

**this**.name = name;

}

}

**public** **class** demo

{

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

{

Consumer<Movie> c1=m->System.***out***.println(m.name+"ready to release");

Consumer<Movie> c2=m->System.***out***.println(m.name+"but flop");

Consumer<Movie> c3=m->System.***out***.println(m.name+"info stored");

Consumer<Movie> cc=c1.andThen(c2).andThen(c3);

Movie m=**new** Movie("Spyder");

cc.accept(m);

}

}

Op

Spyderready to release

Spyderbut flop

Spyderinfo stored

Take some input and perform some conditional checks and return Boolean value->> predicate

Take some input and perform some operation and return any type->> consumer

Take some input and perform required operation and not return anything->> consumer

**Supplier:**

No input just supply my required object

Interface Supplier<R>

{

Public R get();

}

**import** java.util.Date;

**import** java.util.function.\*;

**public** **class** demo

{

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

{

Supplier<Date> s=()->**new** Date();

System.***out***.println(s.get());

}

}

o/p

Thu May 02 11:16:35 IST 2024

**Supplier for random otp**

**import** java.util.Date;

**import** java.util.function.\*;

**public** **class** demo

{

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

{

Supplier<String> s=()->{

String otp="";

**for** (**int** i=0;i<6;i++)

{

otp=otp+(**int**)(Math.*random*()\*10);

}

**return** otp;

};

System.***out***.println(s.get());

}

}

Op

007651

Random() will give value within >=0 and <1

**import** java.util.Date;

**import** java.util.function.\*;

**public** **class** demo

{

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

{

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

System.***out***.println(p.test(4,6));

}

}

true

**BiPredicate**:

Predicate,function and consumer takes only one argument . for 2 input we should go for Bipredicate or BiFunction or BiConsumer

Interface BiPredicate(T1 t1,T2 t2)

{

Public Boolean test(T1 t1,T2 t2);

}

**BiFunction:**

Interface BiFunction<T,U,R>()

{

Public R apply(T t,U u);

}

**import** java.util.ArrayList;

**import** java.util.Date;

**import** java.util.function.\*;

**class** Employee

{

String name;

**int** eno;

**public** Employee(String name, **int** eno)

{

**this**.name = name;

**this**.eno = eno;

}

}

**public** **class** demo

{

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

{

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

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

l.add(f.apply("saswat", 1));

l.add(f.apply("rakesh", 2));

l.add(f.apply("manas", 3));

**for** (Employee e: l)

{

System.***out***.println("employee name "+e.name);

System.***out***.println("employee number "+e.eno);

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

}

}

}

employee name saswat

employee number 1

employee name rakesh

employee number 2

employee name manas

employee number 3\

**BiConsumer:**

**import** java.util.ArrayList;

**import** java.util.Date;

**import** java.util.function.\*;

**class** Employee

{

String name;

**double** salary;

**public** Employee(String name, **double** salary)

{

**this**.name = name;

**this**.salary= salary;

}

}

**public** **class** demo

{

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

{

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

*populate*(l);

BiConsumer<Employee,Double> c=(e,d)->e.salary=e.salary+d;

**for** (Employee e:l)

{

c.accept(e,500.0);

}

**for** (Employee e:l)

{

System.***out***.println("employee name: "+e.name);

System.***out***.println("employee salary: "+e.salary);

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

}

}

**public** **static** **void** populate(ArrayList<Employee> l)

{

l.add(**new** Employee("saswat",1000));

l.add(**new** Employee("rakesh",3000));

l.add(**new** Employee("dipak",5000));

}

}

employee name: saswat

employee salary: 1500.0

employee name: rakesh

employee salary: 3500.0

employee name: dipak

employee salary: 5500.0

if we are providing any int value to predicate will convert into Integer object then converting to int which is called autoboxing and auto unboxing.

This will cause performance issue . to overcome this we have IntPredicate.

**Primitive predicate type**

IntPredicate

DoublePredicate

LongPredicate

**Primitive function types**

DoubleFunction-takes double as argument and return can be anything.

IntFunction- takes int as argument and return can be anything.

longFuction- takes long as argument and return can be anything.

<https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>

**Method and constructor reference**

Its alternative to lambda expressions.

**public** **class** demo

{

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

{

Runnable r=()->{

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

{

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

}

};

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

t.start();

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

{

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

}

}

}

Instead of LE we can use method reference like below.

**public** **class** demo

{

**public** **static** **void** m1()

{

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

{

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

}

}

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

{

Runnable r=demo::*m1*;

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

t.start();

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

{

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

}

}

}

Method can be nonstatic also but object ref is required for this like below.

Double colon is used as refernciating .here run method is replaced by m1().

**public** **class** demo

{

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

{

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

{

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

}

}

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

{

demo d=**new** demo();

Runnable r=d::m1;

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

t.start();

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

{

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

}

}

}

For static:- classname::method name

For non-static:- obj ref::method name

If we have existing method we can use method ref instead of LE.

Arguments should be same remaining things can be anything.

Public void run();

Public void m1();we can use int as return type and different modifier.

Different return types are allowed in method reference.

If implementation is available we should go for method references if there is no impl we should go for LE.

**interface** InterF

{

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

}

**class** Test

{

**public** **static** **void** sum(**int** x,**int** y)

{

System.***out***.println(x+y);

}

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

{

InterF i=(a,b)->System.***out***.println(a+b);

i.add(10,20);

InterF i1=Test::*sum*;

i1.add(30, 40);

}

}

o/p  
30

70

**Constructor reference:**

classname::new

when FI method return an object then this is required.

**class** Sample

{

Sample()

{

System.***out***.println("sample class const called");

}

}

**interface** InterF

{

**public** Sample get();

}

**class** Test

{

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

{

InterF i=Sample::**new**;

Sample s1=i.get();

}

}

sample class const called

in above, interface get() refers Sample class constructor.

If there is a argument then constructor should have same signature.

**class** Sample

{

Sample(String s)

{

System.***out***.println("sample class const called"+s);

}

}

**interface** InterF

{

**public** Sample get(String s);

}

**class** Test

{

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

{

InterF i=Sample::**new**;

Sample s1=i.get("saswat");

}

}

o/p:- sample class const calledsaswat

the advantage of methods ref and constructor ref is code reusability.

If there are multiple constructor then it is always look for matched argument constructor.

It’s an alternative for LE or normal method impl.

Object is being created internally on above.

**Streams:-**

Representing group of single entity is called collection. To process objects from collection then streams are required.

Diff:- IOStream refers stream of binary data collection stream refers stream of objects.

Stream s= c.stream();//returns an stream objects

Present on java.util package

1.intermediate operations:

Transform stream into another stream.

Filter,map,distinct,sorted,limit etc.

2.terminal operations:

It produce the result and terminate the stream.

Eg:- forEach,collect,reduce.count etc.

Filter():-

It filter the elements in stream based on some condition.

Takes argument which returns Boolean type.

Stream filteredStream=originalStream.filter(element->predicate);

**class** Test

{

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

{

List<Integer> list=Arrays.*asList*(5,10,20,25,45);

List<Integer> l1=list.stream().filter(i->i%2==0).collect(Collectors.*toList*());

//or

List<Integer> l2=list.stream().filter(i->i%2==0).toList();

System.***out***.println(l1);

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

}

}

[10, 20]

[10, 20]

Map():

Used to transform the element of stream and returns new stream.

Takes input function as a argument ,return type based on type of data.

Stream mappededStream=originalStream.filter(element-> transformationfunction);

**class** Test

{

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

{

List<Integer> list=Arrays.*asList*(5,10,20,25,45);

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

List<Integer> l2=list.stream().map(i->i\*2).toList();

System.***out***.println(l1);

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

}

}

[10, 20, 40, 50, 90]

[10, 20, 40, 50, 90]

**class** Test

{

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

{

List<Integer> marks=Arrays.*asList*(5,10,20,25,45);

List<Integer> passed=marks.stream().filter(i->i>35).toList();

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

List<Integer> grace=marks.stream().filter(i->i<35).map(j->j+5).toList();

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

}

}

[45]

[10, 15, 25, 30]

Count():

To count the number of element of a stream.

**class** Test

{

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

{

List<Integer> marks=Arrays.*asList*(5,10,20,25,45);

Long failed=marks.stream().filter(i->i<35).count();

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

}

}

4

Sorted():

Sort element of stream based on natural sorting order.

**class** Test

{

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

{

List<Integer> marks=Arrays.*asList*(5,10,20,25,45);

List<Integer> sorted=marks.stream().sorted().toList();

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

}

}

[5, 10, 20, 25, 45]

For customize sorting order we should go for sorted(Comparator c)

**class** Test

{

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

{

List<Integer> marks=Arrays.*asList*(5,10,20,25,45);

List<Integer> descsorted=marks.stream().sorted((a,b)->a<b?1:a>b?-1:0).toList();

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

}

}

[45, 25, 20, 10, 5]

**class** Test

{

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

{

List<Integer> marks=Arrays.*asList*(5,10,20,25,45);

List<Integer> descsorted=marks.stream().sorted((a,b)->b.compareTo(a)).toList();

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

}

}

[45, 25, 20, 10, 5]

**class** Test

{

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

{

List<Integer> marks=Arrays.*asList*(5,10,20,25,45);

List<Integer> descsorted=marks.stream().sorted(Comparator.*reverseOrder*()).toList();

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

}

}

[45, 25, 20, 10, 5]

**class** Test

{

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

{

List<Integer> marks=Arrays.*asList*(5,10,20,25,45);

List<Integer> descsorted=marks.stream().sorted((a,b)->-a.compareTo(b)).toList();

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

}

}

[45, 25, 20, 10, 5]