**Java 8:**

Came on 2014 march 18th

Advantages:

Concise code

Enabling functional programming

**Lambda Expressions**

The main objective to get benefit of functional interface.

It’s an anonymous(nameless) function.

**Syntax of anonymous(nameless) function.:**

Nameless

Without return type

Without modifier

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

{

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

}

Convert into LE

()->{System.***out***.println("hello");}//body of LE is System.***out***.println("hello");

Or

()->System.***out***.println("hello");

Curly braces are optional . if multiple line are present LE body then curly braces are mandatory.

**public** **void** m1(**int** a, **int** b)

{

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

}

Convert to LE

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

Compiler can guess the type automatically.

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

**public** **int** square(**int** n)

{

return n\*n);

}

(**int** n)->{ return n\*n};//we can use return keyword within curly braces only

(n)->n\*n//return is being present automatically and type will be considered as int automatically.

If we have only one input argument then no need of parenthesis.

n->n\*n;

**public** **void** m1(String s)

{

**return** s.length();

}

s->s.length();

n-return n\*n;//invalid//curly braces missing

n->{return n\*n;};//valid

n->{return n\*n};//invalid//semicolon is missing

n->{n\*n;};//invalid//return missing

n->n\*n;//valid

without curly braces we can’t use return keyword.

Within curly braces if we want to return some value then compulsory we should use return statement.

**Functional Interface**

If a interface contains only one abstract method then it is a FI.

It can contain any number of default method and static method but abstract method should be one.

@functionalInterface

@FunctionalInterface//it’s optional

**interface** InterF

{

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

**default** **void** m2()

{}

**public** **static** **void** m3()

{}

}

@FunctionalInterface

**interface** InterF

{

**default** **void** m2()

{}

**public** **static** **void** m3()

{}

}

CTError// it’s expecting abstract method

@FunctionalInterface

**interface** A

{

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

}

@FunctionalInterface

**interface** B **extends** A

{

}

Above is valid because parent method is available on child also. Since child interface contains only one abstract method then it is valid.

@FunctionalInterface

**interface** A

{

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

}

@FunctionalInterface

**interface** B **extends** A

{

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

}

Valid because child is overriding.

@FunctionalInterface

**interface** A

{

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

}

@FunctionalInterface

**interface** B **extends** A

{

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

}

Invalid because child contains two abstract method.

@FunctionalInterface

**interface** InterF

{

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

}

**class** Demo **implements** InterF

{

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

{

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

}

**class** Test

{

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

{

Demo d=**new** Demo();

or

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

i.m1();

}

}

Convert by LE

@FunctionalInterface

**interface** InterF

{

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

}

**class** Test

{

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

{

InterF i=()->System.***out***.println("helllo");

i.m1();

}

}

To call FI method then LE is required.

@FunctionalInterface

**interface** InterF

{

**public** **void** m1(**int** a,**int** b);

}

**class** Test

{

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

{

InterF i=(a,b)->System.***out***.println(a+b);//(a,b) refers to add method

i.m1(10,20);

}

}

If the complier doing all automatically then there will no be performance issue because performance is related to run time not to compile time.

**class** Test

{

**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");

}

}

}

2 threads are running now. Runnable class can be implemented by LE since Runnable is a FI.

**class** Test

{

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

{

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

a.add(30);

a.add(40);

a.add(50);

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

Collections.*sort*(a,c);

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

}

}

**Anonymous inner class vs LE**

A class which has no name then it can be called as AIC.

Wherever functionality is required there only we can provide implementation rather to extend it and provide impl there.

**class** Test

{

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

{

Runnable r= **new** Runnable()

{

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

{

**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");

}

}

}

AIC can be used for any interface but LE can be used for FI only.

**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 static 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

demo.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

or

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]

**class** Test

{

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

{

List<String> list=Arrays.*asList*("A","AAA","BB","BBBBBB");

Comparator<String> c= (a,b)->

{

**int** i1=a.length();

**int** i2=b.length();

**if** (i1>i2)**return** 1;

**else** **if**(i1<i2)

**return** -1;

**else** **return** 0;

};

List<String> sorted=list.stream().sorted(c).toList();

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

}

}

[A, BB, AAA, BBBBBB]

**Min() and max():**

Min(Compartor): return 1st element of list filtered by comparator.

max(Compartor): return last element of list filtered by comparator.

**class** Test

{

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

{

List<String> list=Arrays.*asList*("A","AAA","BB","BBBBBB");

Comparator<String> c= (a,b)->

{

**int** i1=a.length();

**int** i2=b.length();

**if** (i1>i2)**return** 1;

**else** **if**(i1<i2)

**return** -1;

**else** **return** 0;

};

String sorted=list.stream().min(c).get();

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

}

}

A

**forEach():**

to perform the action for each element of the stream.

It’s a terminal operation.

List.stream().forEach(Consumer);

**class** Test

{

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

{

List<String> list=Arrays.*asList*("A","AAA","BB","BBBBBB");

list.stream().forEach(i->System.***out***.println(i));

}

}

A

AAA

BB

BBBBBB

**class** Test

{

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

{

List<String> list=Arrays.*asList*("A","AAA","BB","BBBBBB");

list.forEach(System.***out***::println);

}

}

A

AAA

BB

BBBBBB

**toArray():**

Convert stream object into array

Terminal operation

Stream.toArray()

**class** Test

{

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

{

List<String> list=Arrays.*asList*("A","AAA","BB","BBBBBB");

String[] s=(String[]) list.stream().toArray();

}

}

Convert array to stream

Arrays.stream(arr) Stream.of(arr)

**Stream.of(args):**

Argument can be any type either array or group of elements.

**class** Test

{

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

{

Integer[] a= {10,15,40,60};

Arrays.*stream*(a).filter(i->i%2==0).forEach(System.***out***::println);

}

}

10

40

60

**class** Test

{

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

{

Stream<?> item =Stream.*of*(9,8,"AA");

item.forEach(System.***out***::println);

}

}

9

8

AA

Employees of max salary

**class** Employee

{

**private** String name;

**private** Double salary;

**private** Date joiningDate;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** Double getSalary() {

**return** salary;

}

**public** **void** setSalary(Double salary) {

**this**.salary = salary;

}

**public** Date getJoiningDate() {

**return** joiningDate;

}

**public** **void** setJoiningDate(Date joiningDate) {

**this**.joiningDate = joiningDate;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**private** String gender;

**public** Employee(String name, Double salary, Date joiningDate, String gender) {

**super**();

**this**.name = name;

**this**.salary = salary;

**this**.joiningDate = joiningDate;

**this**.gender = gender;

}

@Override

**public** String toString() {

**return** "Employee [name=" + name + ", salary=" + salary + ", joiningDate=" + joiningDate + ", gender=" + gender

+ "]";

}

}

**class** Test

{

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

{

SimpleDateFormat dateFormat=**new** SimpleDateFormat("dd-mm-yyyy");

Date joiningDateRam=dateFormat.parse("01-01-2000");

Date joiningDateSam=dateFormat.parse("08-11-1998");

Date joiningDateDev=dateFormat.parse("04-07-2000");

Date joiningDateSita=dateFormat.parse("03-05-2002");

List<Employee> emp=Arrays.*asList*(

**new** Employee("ram",20000.0,joiningDateRam,"male"),

**new** Employee("sam",30000.0,joiningDateSam,"male"),

**new** Employee("dev",50000.0,joiningDateDev,"male"),

**new** Employee("sita",90000.0,joiningDateRam,"female"));

Employee maxSalary=emp.stream().max((a,b)->Double.*compare*(a.getSalary(), b.getSalary())).get();

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

}

}

Second highest salary

Optional<Employee> listEmp=emp.stream().

sorted((a,b)->Double.*compare*(b.getSalary(), a.getSalary()))

.skip(1).findFirst();

listEmp.ifPresent(System.***out***::println);

senior employee based on joining date

Optional<Employee> listEmp=emp.stream().

min((a,b)->a.getJoiningDate().compareTo(b.getJoiningDate()));

listEmp.ifPresent(System.***out***::println);

count employees based on gender

Map<String,Long> listEmp=emp.stream().

collect(Collectors.*groupingBy*((Employee::getGender),Collectors.*counting*()));

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

{female=1, male=3}

**Date & Time API(joda-time-api)**

Developed by joda.org

We have various classes for Date and time on 1.7v but for increase performance new date time api introduced.

Import java.time.\*;

LocalDate date=LocalDate.now();

Sop(date);

LocalTime time=LocalTime.now();

Sop(time);

LocalDate date=LocalDate.now();

Int dd=date.getDayOfMonth();

Int mm= date.getMonthValue();

Int yyyy=date.getYear();

SoprintF(“%d-%d-%d”,dd,mm,yyyy);----prints in dd-mm-yyyy format

LocalTime time=LocalTime.now();

Int h=time.getHour();

Int m=time.getMinute();

Int s=time.getSecond();

Int n=time.getNano();

SoprintF(“%d:%d:%d:%d”,h,m,s,n);---- prints in hh:mm:sec:nanosec format

LocalDateTime dt=LocalDateTime.now();

Sop(dt);////gives date and time

LocalDateTime dt=LocalDateTime.now();

Int dd=de.getDayOfMonth();

Int mm=dt.getMonthValue();

Int yy=dt.getYear();

Int h=dt.getHour();

Int m=dt.getMinute();

Int s=dt.getSecond();

Int n=dt.getNano();

LocalDateTime dt=LocalDateTime.of(yy,mm,dd,h,m,s,ns);

Upto how much values we know we can ether that.

LocalDateTime dt=LocalDateTime.of(1995,05,26,12,40);

LocalDateTime dt=LocalDateTime.of(1995,Month.may,26,12,40);///may is a constant inside Month class

Sop(dt);///1995-05-26T12:40

LocalDateTime dt=LocalDateTime.of(1995,05,26,12,40);

Sop(dt.plusMonths(6));

Sop(dt.minusMonths(6));

**Period:-**

LocalDate birthday=LocalDate.of(1998,11,08);

LocalDate today=LocalDate.now();

Period p=Period.between(birthday,today);

Sopf(“age is %d years %d months %d days”,p.getYears(),p.getMonths(),p.getDays());

LocalDate deathday= LocalDate.of(1998+60,11,08);

Period p1=Period.between(today,deathday);

Int d= p1.getYears()\*365+p1.getMonths()\*30+p1.getDays();

**Year**

Scanner sc=new Scanner(System.in);

Sop(“enter year”);

Int n= sc.nextInt();

Year y= Year.of(n);

If(y.isLeap())

{

Sop(y+“is leap year”);

}

Else

{

Sop(y+“is not leap year”);

}

**ZoneId**

It’s a class to represent zone.

ZoneId zone=ZoneId.systemDefault();//asia/calcuttta

ZoneId la =ZoneId.of(“America/los-angels”);

ZonedDateTime zt=ZonedDateTime.now(la);//gives current time of los angels.

**public** **class** **Main** {

**public** **static** void printDuplicates(String str)

{

int len = str.length();

*// Sorting the string*

char[] chars = str.toCharArray();

Arrays.sort(chars);

String sortedStr = **new** String(chars);

*// Loop through the sorted string to find duplicates*

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

int count = 1;

*// Counting the occurrences of each character*

**while** (i < len - 1

&& sortedStr.charAt(i)

== sortedStr.charAt(i + 1)) {

count++;

i++;

}

*// Printing the duplicate character and its*

*// count*

**if** (count > 1) {

System.out.println(sortedStr.charAt(i)

+ ", count = " + count);

}

}

}

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

{

String str = "test string";

printDuplicates(str);

}

}

**public** **class** DuplicateCharacters {

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

String string1 = "Great responsibility";

**int** count;

//Converts given string into character array

**char** string[] = string1.toCharArray();

System.***out***.println("Duplicate characters in a given string: ");

//Counts each character present in the string

**for**(**int** i = 0; i <string.length; i++)

{

count = 1;

**for**(**int** j = i+1; j <string.length; j++)

{

**if**(string[i] == string[j] && string[i] != ' ')

{

count++;

//Set string[j] to 0 to avoid printing visited character

string[j] = '0';

}

}

//A character is considered as duplicate if count is greater than 1

**if**(count > 1 && string[i] != '0')

System.***out***.println(string[i]);

}

}

}