Java 8

**It was introduced in March 18 2014.**

**In this version Java add lot of features was added.**

**Those are:**

1. **Lambda Expressions.**
2. **Method References.**
3. **Functional interface.**
4. **Stream API.**
5. **Base64 encode and decode.**
6. **Default Methods interface.**
7. **Static methods in interface**
8. **forEach() method.**
9. **Collectors class.**
10. **StringJoiner class.**
11. **Optional Class.**
12. **JavaScript Nashorn.**
13. **Parllel array sort.**
14. **Parameter reflection.**
15. **Type annotations.**
16. **LocalDate,LocalTime,LocalDateTime,Period,Year**
17. **Type inference**

**Types of Interfaces:**

**From JDK 1.8 we have three types of interfaces.**

1. **Normal interface.**
2. **Marker interface or tagling interface.**
3. **Functional interface.**

**Normal Interface:**

**An interface, which contains one or multiple abstract method.**

**Interface I{**

**Public abstract void m1();**

**}**

**Interface J{**

**Public abstract void m1();**

**Public abstract void m2();**

**}**

**Marker Interface:**

**An interface, which doesn’t contains any one abstract method.**

**Interface I{**

**}**

**Functional Interface:**

**An interface, which contains exactly only one method.**

**It is always annotated with @FunctionalInterface annotation.**

**@FunctionalInterface**

**Interface I{**

**Public abstract void m1();**

**}**

**Lambda Expressions:**

**It is very important feature added in java 8.**

**Java doesn’t support functional programming up to java 7, but from java 8 onwards java is also supports functional programming.**

**It is useful for developing anonymous functions in java.**

**It is best suitable for providing implementation for Single abstract method interface/functional interface/one method interface.**

**It will provide implementation for functional interface in the very short format and clear manner.**

**It is best suitable for collection framework library.**

**It is the best suitable for navigating/iterating, filtering and read or extract the data from collection objects.**

**Before talk about Lambda expression first we must and should be aware about Functional Interface.**

**An interface which contains exactly only one abstract method is called Functional Interface or Single abstract method interface or single/one method interface.**

**To represents an interface as Functional Interface, we have one annotation like @FunctionalInterface.**

**Lambda is an object.**

**DIFFERENCES BETWEEN ANONYMOUS INNER CLASS AND LAMBDA EXPRESSION:**

**Both are provides implementation for interface.**

**Anonymous inner class doesn’t have name.**

**Anonymous inner class creates the .class file.**

**Anonymous inner class is suitable for providing implementations for all type of interfaces(marker, functional, general).**

**interface MI{**

**}**

**@FunctionalInterface**

**interface FI{**

**public abstract void m1();**

**}**

**interface GI{**

**public abstract void m2();**

**public abstract void m3();**

**}**

**class Test{**

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

**MI obj1 = new MI()**

**{**

**};**

**FI obj2 = new FI()**

**{**

**public void m1(){**

**System.out.println("m1 method-FI-AIC ");**

**}**

**};**

**obj2.m1();**

**GI obj3 = new GI()**

**{**

**public void m2(){**

**System.out.println("m2 method-GI-AIC");**

**}**

**public void m3(){**

**System.out.println("m3 method-GI-AIC");**

**}**

**};**

**obj3.m2();**

**obj3.m3();**

**}**

**}**

**Lambda expression is an anonymous expression or function(no name) which will provide implementation for Functional interface or general interface but which contains only one method.**

**Lambda expression doesn’t create .class file.**

**How to provide implementation for interface up to java 7?**

1. **With the support of implementation class.**

**Example:**

**interface** Test{

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

}

**class** Check **implements** Test{

@Override

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

System.***out***.println("This is implementation method for Test interface.....");

}

}

**public** **class** Lambda {

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

Test t = **new** Check();

t.m1();

}

}

1. **We have weapon like anonymous inner class to provide implementation for interfaces (contains only one method) like bellow.**

**Example:**

**interface** Test{

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

}

**public** **class** Lambda {

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

Test t = **new** Test() {

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

System.***out***.println("This is the implementation for Test interface...");

}

};

t.m1();

}

}

1. **We have weapon like anonymous inner class to provide implementation for interfaces (contains more than one method) like bellow.**

**Example:**

**interface** Test{

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

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

}

**public** **class** Lambda {

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

Test t = **new** Test() {

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

System.***out***.println("This is the implementation for Test interface...m1()");

}

**public** **void** m2() {

System.***out***.println("This is the implementation for Test interface...m2()");

}

};

t.m1();

t.m2();

}

}

**In the above program we are getting bellow .class files**

1. **Test.class**
2. **Lambda.class**
3. **Lambda$1.class**

**Every anonymous class is the sub or implementation class for either class or interface.**

1. **From java 8 onwards we have one greater weapon that is LAMBDA EXPRESSION.**

**It is suitable for Single Abstract Method Interface.**

**interface** Test{

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

}

**public** **class** Lambda {

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

Test t = () -> {

System.***out***.println("this is implementation for Test interface"

+ " by using lamda");

};

t.m1();

}

}

**In the above we are getting only two .class file like**

1. **Test.class**
2. **Lambda.class**

**Here no .class file like Lambda$1.class file.**

**That means in the above we are not writing any class separately. This is the example highlighting functional programming in java.**

**Example on anonymous inner class with zero argument/ non-parameterized method.**

**interface** Square{

**public** **static** **final** **int** ***side***=50;

**public** **abstract** **void** area( );

}

**public** **class** Lambda {

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

Square t = **new** Square() {

**public** **void** area() {

System.***out***.println("The area of the square is:: "+(4\****side***));

}

};

t.area();

}

}

**In the above program “side” not available in anonymous inner class but it will takes from interface.**

**Example on Lambda Expression for Single Abstract method interface with zero argument/non-parameterized method.**

**interface** Square{

**public** **static** **final** **int** ***side***=50;

**public** **abstract** **void** area();

}

**public** **class** Lambda {

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

Square t =() -> {

System.***out***.println("The area of the square is:: "+(4\*side));

};

t.area();

}

}

**We will get compile time error. That is can’t find symbol variable side.**

**In the above side variable is not taking from the Square interface the reason lambda is not an implementation class of Square. It is a just function.**

**To overcome the above problem, take one local “side” variable in main method like bellow.**

**interface** Square{

**public** **abstract** **void** area();

}

**public** **class** Lambda {

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

**int** side=500;

Square t =() -> {

System.***out***.println("The area of the square is:: "+(4\*side));

};

t.area();

}

}

**We can take “side” variable as static at class level like bellow.**

**interface** Square{

**public** **abstract** **void** area();

}

**public** **class** Lambda {

**static** **int** *side*=14;

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

Square t =() -> {

System.***out***.println("The area of the square is:: "+(4\**side*));

};

t.area();

}

}

**But don’t take “side” variable as non-static/instance the reason is non-static data we can’t call directly from static context.**

**interface** Square{

**public** **abstract** **void** area();

}

**public** **class** Lambda {

**int** side=14;

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

Square t =() -> {

System.***out***.println("The area of the square is:: "+(4\*side));

};

t.area();

}

}

**Compile Time Error: non-static variable cannot be referenced from static context.**

**Program on Lambda Expression of Functional interface which contains argument method.**

**interface** Square{

**public** **abstract** **void** area(**int** side);

}

**public** **class** Lambda {

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

Square t = (side) -> {

System.***out***.println("The area of the square is:: "+(4\*side));

};

t.area (111);

}

}

**Program on Lambda Expression of Functional interface which contains argument method.**

**interface** Rectangle{

**public** **abstract** **void** area(**int** length, **int** breadth);

}

**public** **class** Lambda {

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

Rectangle t =(a,b) -> {

System.***out***.println("The area of the square is:: "+(a\*b));

};

t.area(10,20);

}

}

**Program on Lambda Expression of functional interface which contains non-void method.**

**In Java if the method is non-void method, that method must and should be ended with return with appropriate value.**

**In functional programming (Lambda Expression), no need to write return keyword.**

**Example:**

**interface** Rectangle{

**public** **abstract** **int** area(**int** length, **int** breadth);

}

**public** **class** Lambda {

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

Rectangle t =(a,b) -> {

**return** a\*b;

};

System.***out***.println(t.area(100,200));

Rectangle t1 =(a,b)->(2\*(a+b));

System.***out***.println(t1.area(10,20));

}

}

**In the last but one line don’t try to write body like bellow.**

**Rectangle t1 = (a, b) ->{(2\*(a+b))};**

**(Or)**

**Rectangle t1 = (a, b) ->{(2\*(a+b));};**

**If we trying to write body we need write return keyword after curly braces.**

**Rectangle t1 =(a,b)->{return 2\*(a+b);};**

**System.*out*.println(t1.area(10,20));**

**How to trace out Collection Object elements without Lambda Expression:**

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**public** **class** Lambda {

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

List<String> l = **new** ArrayList<String>();

l.add("nit");

l.add("kit");

l.add("nacre");

l.add("ram");

l.add("cj");

l.add("aj");

Iterator i = l.iterator();

**while**(i.hasNext()) {

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

}

}

}

**How to trace out Collection Object elements with Lambda Expression:**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Lambda {

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

List<String> l = **new** ArrayList<String>();

l.add("nit");

l.add("kit");

l.add("nacre");

l.add("ram");

l.add("cj");

l.add("aj");

l.forEach((data)->{System.***out***.println(data);});

}

}

**How to trace out Collection Object elements with Lambda Expression:**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Lambda {

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

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

l.add(10);

l.add(20);

l.add(30);

l.add(40);

l.add(50);

l.add(60);

l.forEach((I)->{System.***out***.println(I);});

}

}

**Program on Lambda Expression with multiple statements in body:**

**interface** Rectangle{

**public** **abstract** **void** test(**int** length, **int** breadth);

}

**public** **class** Lambda {

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

Rectangle s = (length,breadth)->{

**int** perimeter = 2\*(length+breadth);

System.***out***.println("perimeter: "+perimeter);

**int** area = (length\*breadth);

System.***out***.println("area: "+area);

};

s.test(100, 200);

}

}

**How to create Thread by using Runnable interface:**

**class** Test **implements** Runnable{

@Override

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

System.***out***.println("This Test class run method");

}

}

**public** **class** Lambda {

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

Test t = **new** Test();

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

tt.start();

}

}

**How to create Thread by using anonymous inner class:**

**public** **class** Lambda {

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

Runnable r = **new** Runnable()

{

@Override

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

System.***out***.println("This is thread: anonymous inner class");

}

};

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

t.start();

}

}

**How to create Thread by using Lambda Expression:**

**public** **class** Lambda {

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

Runnable r = ()->{

System.***out***.println("This is thread: Lamda Expression");

};

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

t.start();

}

}

**How to create userdefine threads by using anonymous inner class and lambdas:**

* **Excecuting logic of run() by UDT**
* **Excecuting logic of other method by UDT**

**package** java8newfeatures;

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

**public** **static** **void** main(String []x) **throws** InterruptedException{

//anonymous inner class is either sub class of another class or implementation class of interface

Thread t = **new** Thread()

{//anonymous inner class it is sub class of java.lang.Thread

@Override

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

System.***out***.println("run: "+Thread.*currentThread*().getName());

display();

}

**void** display(){

System.***out***.println("display: "+

Thread.*currentThread*().getName());

}

};

t.start();

Runnable r = **new** Runnable()

{//anonymous inner class, it is the implementation class of Runnable interface

@Override

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

System.***out***.println("run: "+Thread.*currentThread*().getName());

mm1();

}

**void** mm1(){

System.***out***.println("mm1: "+Thread.*currentThread*().getName());

}

};

Thread t1 = **new** Thread(r); t1.start();

Thread.*sleep*(1000);

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

//Thread tt = () -> {}; //we cant apply lambda on classes

Runnable r1 = () -> {

System.***out***.println("LAMBDAS EXPRESSION");

System.***out***.println("run: "+Thread.*currentThread*().getName());

Test.*mm2*();

Test t5 = **new** Test();

t5.mm3();

};

Thread tt1 = **new** Thread(r1);

tt1.start();

}

**static** **void** mm2(){

System.***out***.println("mm2: "+Thread.*currentThread*().getName());

}

**void** mm3(){

System.***out***.println("mm3: "+Thread.*currentThread*().getName());

}

}

**How to create userdefine theread by implements java.util.concurrent.Callable interface by using both inner classes and lambda expression.**

**package** java8newfeatures;

**import** java.util.concurrent.Callable;

**import** java.util.concurrent.ExecutionException;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.Future;

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

**public** **static** **void** main(String []x) **throws** InterruptedException, ExecutionException{

Callable c = **new** Callable()

{

@Override

**public** Object call(){

System.***out***.println("call: "+Thread.*currentThread*().getName());

**int** result=0;

**for**(**int** i=1;i<=5;i++)

result = result+i;

**return** result;

}

};

ExecutorService cs = Executors.*newFixedThreadPool*(2);

Future f = cs.submit(c);

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

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

Callable c1 = () ->

{

System.***out***.println("call: "+Thread.*currentThread*().getName());

**int** result=0;

**for**(**int** i=1;i<=5;i++)

result = result+i;

**return** result;

};

ExecutorService cs1 = Executors.*newFixedThreadPool*(1);

Future f1 = cs1.submit(c);

System.***out***.println(f1.get());

}

}

**How to sort the elements on Collection Object by using comparator without using Comparator:**

**import** java.util.Comparator;

**import** java.util.Iterator;

**import** java.util.Set;

**import** java.util.TreeSet;

**import** oracle.net.aso.e;

**class** Student{

**int** sid;

String sname;

Integer sage;

**public** Student(**int** sid, String sname, **int** sage) {

**super**();

**this**.sid = sid;

**this**.sname = sname;

**this**.sage = sage;

}

@Override

**public** String toString() {

**return** sid+"..."+sname+"..."+sage;

}

}

**public** **class** Lambda {

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

Student s1 = **new** Student(101,"ram",30);

Student s2 = **new** Student(102,"sam",20);

Student s3 = **new** Student(103,"varun",40);

Student s4 = **new** Student(104,"kiran",50);

Student s5 = **new** Student(105,"uma",10);

Set<Student> l = **new** TreeSet<Student>(**new** MyComparator());

l.add(s1);

l.add(s2);

l.add(s3);

l.add(s4);

l.add(s5);

Iterator<Student> i = l.iterator();

**while**(i.hasNext()) {

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

}

System.***out***.println("program finished");

}

}

**class** MyComparator **implements** Comparator{

@Override

**public** **int** compare(Object obj1, Object obj2) {

Student s1 = (Student)obj1;

Student s2 = (Student)obj2;

**return** -s2.sage.compareTo(s1.sage);

}

}

**Lambda Expression on Collection Object sorting elements based Student name:**

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**class** Student{

**int** sid;

String sname;

**int** sage;

**public** Student(**int** sid, String sname, **int** sage) {

**super**();

**this**.sid = sid;

**this**.sname = sname;

**this**.sage = sage;

}

@Override

**public** String toString() {

**return** sid+"..."+sname+"..."+sage;

}

}

**public** **class** Lambda {

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

Student s1 = **new** Student(101,"ram",30);

Student s2 = **new** Student(102,"sam",20);

Student s3 = **new** Student(103,"varun",40);

Student s4 = **new** Student(104,"kiran",50);

Student s5 = **new** Student(105,"uma",10);

List<Student> l = **new** ArrayList<Student>();

l.add(s1);

l.add(s2);

l.add(s3);

l.add(s4);

l.add(s5);

Collections.*sort*(l, (p1 ,p2)->{

**return** p1.sname.compareTo(p2.sname);

});

l.forEach((data)->{

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

});

}

}

**Note: Don’t take variable names as private.**

**Lambda Expression on Collection Object for sorting the elements based on student age:**

**Note: Don’t take student age in primitive data type format. We should take wrapper class format.**

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**class** Student{

**int** sid;

String sname;

Integer sage;

**public** Student(**int** sid, String sname, **int** sage) {

**super**();

**this**.sid = sid;

**this**.sname = sname;

**this**.sage = sage;

}

@Override

**public** String toString() {

**return** sid+"..."+sname+"..."+sage;

}

}

**public** **class** Lambda {

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

Student s1 = **new** Student(101,"ram",30);

Student s2 = **new** Student(102,"sam",20);

Student s3 = **new** Student(103,"varun",40);

Student s4 = **new** Student(104,"kiran",50);

Student s5 = **new** Student(105,"uma",10);

List<Student> l = **new** ArrayList<Student>();

l.add(s1);

l.add(s2);

l.add(s3);

l.add(s4);

l.add(s5);

Collections.*sort*(l, (p1 ,p2)->{

**return** p1.sage.compareTo(p2.sage);

});

l.forEach((data)->{

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

});

}

}

**How to use Lambda Expression on Filter Object.**

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Stream;

**class** Student{

**int** sid;

String sname;

Integer sage;

**public** Student(**int** sid, String sname, **int** sage) {

**super**();

**this**.sid = sid;

**this**.sname = sname;

**this**.sage = sage;

}

@Override

**public** String toString() {

**return** sid+"..."+sname+"..."+sage;

}

}

**public** **class** Lambda {

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

Student s1 = **new** Student(101,"ram",30);

Student s2 = **new** Student(102,"sam",20);

Student s3 = **new** Student(103,"varun",40);

Student s4 = **new** Student(104,"kiran",50);

Student s5 = **new** Student(105,"uma",10);

List<Student> l = **new** ArrayList<Student>();

l.add(s1);

l.add(s2);

l.add(s3);

l.add(s4);

l.add(s5);

Stream<Student> s= l.stream();

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

Stream<Student> ss = l.stream().filter((std)-> std.sage>15);

ss.forEach((st)->{

System.***out***.println(st.sid+"..."+st.sname+"..."+st.sage);

});

}

}

**Generic Functional Interface:**

**Lambda Expression by default these are not a generic, where as Functional interfaces are generic.**

**Syntax:**

**@FunctionalInterface**

**Interface Interface\_name<Type>{**

**<type>method\_name(<type> o);**

**}**

@FunctionalInterface

**interface** I<T>{

**public** **abstract** T m3(T t);

}

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

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

System.***out***.println("test class");

I<Integer> obj = (Integer t) -> {

**return** t\*t;

};

**int** x = obj.m3(20);

System.***out***.println("x: "+x);

}

}

In the above lambda expression we can write

(Iteger t) ->

Or

(t) ->

**@FunctionalInterface**

**interface FI{**

**public abstract int m1(int x);**

**}**

**@FunctionalInterface**

**interface GFI<T>{**

**public abstract T m2(T x);**

**}**

**public class Test{**

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

**FI obj = new FI()**

**{**

**public int m1(int x){**

**return x\*x;**

**}**

**};**

**int result1 = obj.m1(5);**

**System.*out*.println("result1: "+result1);**

**FI obj1 = (int x) -> {**

**return x\*x;**

**};**

**int result2 = obj1.m1(10);**

**System.*out*.println("result2: "+result2);**

**GFI<Integer> obj2 = new GFI<Integer>()**

**{**

**public Integer m2(Integer x){**

**return x\*x;**

**}**

**};**

**int result3 = obj2.m2(100);**

**System.*out*.println("result3: "+result3);**

**GFI<String> obj3 = new GFI<String>()**

**{**

**public String m2(String x){**

**return "hi "+x;**

**}**

**};**

**String result4 = obj3.m2("ram");**

**System.*out*.println("result4: "+result4);**

**GFI<Integer> obj4 = (Integer x)->{**

**return x\*x;**

**};**

**int result5 = obj4.m2(20);**

**System.*out*.println("result5: "+result5);**

**GFI<String> obj5 = (String ss)->{**

**return "hi mr."+ss;**

**};**

**String result6 = obj5.m2("kiran");**

**System.*out*.println("result6: "+result6);**

**}**

**}**

**Program on how generic functional interface supports to work with different type of objects:**

@FunctionalInterface

**interface** I<T>{

**public** **abstract** T m3(T t);

}

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

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

System.***out***.println("test class");

I<String> obj = (x) ->{

**return** x.length()+"";

};

String y = obj.m3("rams");

System.***out***.println("y: "+y);

I<Integer> obj1 = (Integer i)->{

**return** i\*i;

};

**int** z = obj1.m3(15);

System.***out***.println("z: "+z);

}

}

**What is the difference between block lambda expression and unblock lambda expression:  
In Block Lambda Expression we can able to writ multiple statements, whereas unblock lambda expression we can write only statement.**

**Within the “{ }”, we can write single statements as well as multiple statements, if we write single statement, that statement can be ended with either ‘;’ or we can ignore, but if we write multiple statements, we should separate each and every statements with ‘;’.**

@FunctionalInterface

**interface** I{

**public** **abstract** **int** m1(String s1,String s2);

}

**class** Addition{

**int** add(String s1,String s2){

**int** i = s1.hashCode();

**int** j = s2.hashCode();

**int** k = i+j;

**return** k;

}

}

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

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

I obj = (s1,s2) ->{

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

Addition an = **new** Addition();

**return** an.add(s1,s2);

};

**int** result = obj.m1("Aa", "BB");

System.***out***.println("result: "+result);

}

}

**“this” keyword or reference in lambda expression:**

@FunctionalInterface

**interface** I{

**public** **abstract** String m1();

}

**class** A{

String s1="ram";

I getA(){

String s1="java";

I obj = ()-> {

**return** **this**.s1;

};

**return** obj;

}

}

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

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

A obj = **new** A();

I obj1 = obj.getA();

String value = obj1.m1();

System.***out***.println("value: "+value);

}

}

**Example on lambda including of this and super keywords:**

**@FunctionalInterface**

**interface FI{**

**public abstract int m1();**

**}**

**class B {**

**int a = 444;**

**}**

**class A extends B{**

**int a = 333;**

**FI getObject(){**

**FI obj1 = new FI()**

**{**

**public int m1(){**

**return 111;**

**}**

**};**

**return obj1;**

**}**

**FI getObject1(){**

**int a = 222;**

**FI obj2 = () -> {**

**//return a;**

**//return this.a;**

**return super.a;**

**};**

**return obj2;**

**}**

**}**

**public class Test {**

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

**A obj = new A();**

**FI ref = obj.getObject();**

**System.*out*.println(ref.m1());**

**FI ref1 = obj.getObject1();**

**System.*out*.println(ref1.m1());**

**}**

**}**

**How to forward one function to another function as argument:**

@FunctionalInterface

**interface** I{

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

}

**class** A{

**void** test(I obj){

obj.m1();

}

}

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

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

I obj = () -> {

System.***out***.println("Functional Programming");

};

A obj1 = **new** A();

obj1.test(obj);

}

}

**package** java8newfeatures;

@FunctionalInterface

**interface** I{

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

}

**class** Check{

**static** **void** fp(I obj){

obj.m1();

}

}

**class** A **implements** I{

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

System.***out***.println("m1-A");

}

}

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

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

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

Check.*fp*(**new** A());

I obj1 = **new** I()

{

@Override

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

System.***out***.println("m1-Test$1");

}

};

Check.*fp*(obj1);

I obj2 = ()->{System.***out***.println("m1-lambda");};

Check.*fp*(obj2);

Check.*fp*(()->{System.***out***.println("m1-lambda\*");});

Check.*fp*(Temp::*mm*);

}

}

**class** Temp{

**public** **static** **void** mm(){

System.***out***.println("m1-method reference");

}

}

**Method References:**

**interface** Test{

**public** **abstract** **void** check();

}

**public** **class** Lambda {

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

Test t = () -> {

System.***out***.println("this is test interface check implementation");

};

t.check();

}

}

**In the above program we writing one lambda expression for providing implementation of Test interface check method and later referencing that method.**

**In the above program Lambda Expression may be give a little bit of confusion, to avoiding that confusing and implementation separately in one method and later if we want to pointing that method we have alternative way that is Method References.**

**If we are using Method References internally that method references will write one lambda expression.**

**The representation of Method References is :: (double colon).**

**The above program we can rewrite as bellow by using Method References.**

**interface** Test{

**public** **abstract** **void** check();

}

**public** **class** Lambda {

**public** **static** **void** check() {

System.***out***.println("this is test interface check implementation:Method references");

}

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

Test t = Lambda::*check*;

t.check();

}

}

We can use above method reference on top of following programming elements.

1. Static methods.
2. Non-static/instance methods.
3. Constructors.

**How to create Thread by using lambda expression to call user define method.**

**public** **class** Lambda {

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

System.***out***.println("this is thread executing by: Lambda Expression");;

}

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

Runnable r = ()->*m1*();

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

t.start();

}

}

**How to create Thread by using Method Reference to call user define method.**

**Note: The above program we can rewrite as bellow.**

**public** **class** Lambda {

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

System.***out***.println("this is thread executing by: Method References");;

}

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

Runnable r = Lambda::*m1*;

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

t.start();

}

}

Note: **This statement** Runnable r = Lambda::*m1*;

w**ill internally provide lambda expression like bellow**

**() -> m1();**

**How to communicating with non-static method by using method reference:**

**interface** Test{

**public** **abstract** **void** wish() ;

}

**public** **class** MethodReference {

**public** **void** wish1() {

System.***out***.println("this is thread executing by: Method References");;

}

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

MethodReference mr = **new** MethodReference();

Test t = mr::wish1;

t.wish();

}

}

Example on MethodReference and Lambda Expression to interact with static method:

**import** java.util.function.BiFunction;

**class** Addition{

**public** **static** **int** addition(**int** a,**int** b) {

**return** a+b;

}

}

**public** **class** MethodReference {

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

BiFunction<Integer,Integer,Integer> bi = (a,b)-> (a+b);

**int** i = bi.apply(10, 20);

System.***out***.println("i: "+i);

BiFunction<Integer,Integer,Integer> bi1 = (a,b)->{

**return** (a+b);

};

**int** j = bi.apply(10, 20);

System.***out***.println("j: "+j);

BiFunction<Integer, Integer, Integer> mr = Addition::*addition*;

**int** k = mr.apply(500, 300);

System.***out***.println("k: "+k);

}

}

Example on MethodReference for overloading static methods:

**import** java.util.function.BiFunction;

**class** Addition{

**public** **static** **int** addition(**int** a,**int** b) {

**return** a+b;

}

**public** **static** **float** addition(**int** a,**float** b) {

**return** a+b;

}

**public** **static** String addition(String a,String b) {

**return** a+b;

}

}

**public** **class** MethodReference {

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

BiFunction<Integer, Integer, Integer> mr = Addition::*addition*;

**int** i = mr.apply(500, 300);

System.***out***.println("i: "+i);

BiFunction<Integer, Float, Float> mr1 = Addition::*addition*;

**float** j = mr1.apply(50, 300.00f);

System.***out***.println("j: "+j);

BiFunction<String, String, String> mr2 = Addition::*addition*;

String k = mr2.apply("ram", "chandra");

System.***out***.println("k: "+k);

}

}

Example on MethodReference to communicating with non-static methods:

**import** java.util.function.BiFunction;

**class** Addition{

**public** **int** addition(**int** a,**int** b) {

**return** a+b;

}

**public** **float** addition(**int** a,**float** b) {

**return** a+b;

}

**public** String addition(String a,String b) {

**return** a+b;

}

}

**public** **class** MethodReference {

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

BiFunction<Integer, Integer, Integer> mr = **new** Addition()::addition;

**int** i = mr.apply(500, 300);

System.***out***.println("i: "+i);

BiFunction<Integer, Float, Float> mr1 = **new** Addition()::addition;

**float** j = mr1.apply(50, 300.00f);

System.***out***.println("j: "+j);

BiFunction<String, String, String> mr2 = **new** Addition()::addition;

String k = mr2.apply("ram", "chandra");

System.***out***.println("k: "+k);

}

}

Example on MethodReference to communicating with constructor:  
**interface** Test{

Addition add(**int** a, **int** b);

}

**class** Addition{

Addition(**int** a, **int** b){

System.***out***.println("this is addition class: "+(a+b));

}

}

**public** **class** MethodReference {

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

Test t = Addition::**new**;

t.add(10,20);

}

}

Examples on java.util.function.Predicate:

Example1:

**import** java.util.function.Predicate;

**public** **class** Test1 {

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

Predicate<Integer> p = (x)-> x>5;

**boolean** b = p.test(10);

System.***out***.println("b: "+b);

Predicate<Boolean> p1 = (y)-> y==**true** ;

**boolean** b1 = p1.test(**true**);

System.***out***.println("b1: "+b1);

}

}

Example2:

**import** java.util.ArrayList;

**import** java.util.function.Predicate;

**public** **class** Check{

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

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

Student s1 = **new** Student(101,"ram");

Student s2 = **new** Student(102,"sam");

Student s3 = **new** Student(103,"raj");

al.add(s1);

al.add(s2);

al.add(s3);

Predicate<Student> pp= (Student obj) -> obj.sname.startsWith("s") ;

/\*for(Student s11 : al){

System.out.println(pp.test(s11));

}\*/

al.forEach((x) ->{

**boolean** b = pp.test(x);

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

**if**(b==**true**){

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

}

});

}

}

**class** Student{

**int** sid;

String sname;

Student(**int** sid,String sname){

**this**.sid = sid;

**this**.sname = sname;

}

}

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

**import** java.util.function.Predicate;

**public** **class** Check{

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

/\*Predicate<Integer> p = (x)->

{

if(x >= 0) return true;

else return false;

};

boolean b1 = p.test(10);

boolean b2 = p.test(-10);

System.out.println("b1: "+b1);

System.out.println("b2: "+b2);\*/

Predicate<Integer> p = (i) -> i>=0;

Predicate<Integer> p1 = (j) -> j>=10;

**boolean** result = p.and(p1).test(20);

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

**boolean** result1 = p.and(p1).test(9);

System.***out***.println(result1);

**boolean** result2 = p.or(p1).test(9);

System.***out***.println(result2);

**boolean** result3 = p.negate().test(20);

System.***out***.println("---"+result3);

**boolean** result4 = p.negate().test(-20);

System.***out***.println("---"+result4);

Predicate<String> pp = Predicate.*isEqual*("ram");

**boolean** result5 = pp.test("ram");

//boolean result5 = pp.test("Ram");

System.***out***.println(result5);

Predicate<String> pp1 = Predicate.*isEqual*("ram");

**boolean** result6 = pp1.test(**new** Employee().sname);

System.***out***.println(result6);

Student s1 = **new** Student(101,"ram");

Student s2 = **new** Student(102,"sam");

Student s3 = **new** Student(101,"ram");

Predicate<Student> pp2 = Predicate.*isEqual*(s1);

**boolean** result7 = pp2.test(s2);

System.***out***.println("\*\* "+result7);

**boolean** result8 = pp2.test(s3);

System.***out***.println("\*\* "+result8);

}

}

**class** Employee{

String sname="varun";

}

**class** Student{

**int** sid;

String sname;

Student(**int** sid,String sname){

**this**.sid = sid;

**this**.sname = sname;

}

@Override

**public** **int** hashCode(){

**return** sid+sname.hashCode();

}

@Override

**public** **boolean** equals(Object o){

Student s1 = (Student)o;

**return** **this**.hashCode()==s1.hashCode();

}

}

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

**Program on java.util.function.BiPredicate**

**import** java.util.function.BiPredicate;

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

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

BiPredicate<Integer,Integer> bi1 = (x,y)-> x>y ;

**boolean** b2 = bi1.test(20, 10);

System.***out***.println("b2: "+b2);

BiPredicate<Integer,Integer> bi2 = (x,y)-> x>y ;

**boolean** b3 = bi2.test(10, 20);

System.***out***.println("b3: "+b3);

}

}

**import** java.util.function.BiPredicate;

**public** **class** Check{

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

/\*BiPredicate<Integer,String> bp1 = (x,y)-> (x >= y.length());

//boolean result1 = bp1.test(10, "ramchandr");

boolean result1 = bp1.test(10, "ramchandrarao");

System.out.println("result1: "+result1);\*/

BiPredicate<Integer,String> bp1 = (x,y)-> {

**int** a = Integer.*parseInt*(y);

**int** a1 = a\*a;

**if**(x >= a1){

**return** **true**;

}

**else** **return** **false**;

};

BiPredicate<Integer,String> bp2 = (x,y)-> (x >= y.length());

//boolean b1 = bp1.and(bp2).test(10,"3");//true

//boolean b1 = bp1.and(bp2).test(10,"5");//false

//boolean b1 = bp1.or(bp2).test(10,"3");//true

//boolean b1 = bp1.or(bp2).test(10,"5");//true

**boolean** b1 = bp1.negate().test(10,"5");//true

System.***out***.println("b1: "+b1);

}

}

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

Example on java.util.function.Function:

Example1:

**import** java.util.function.Function;

**public** **class** Test1 {

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

Function<Integer,String> fun = (f) -> "The result is: "+f;

String f1 = fun.apply(10);

System.***out***.println("f1: "+f1);

Function<Integer,String> fun1 = (f) -> {

**if**(f >10 ) {

**return** "Given Number Is: "+f+" greater than 10";

}

**else** **return** "Given Number Is: "+f+" lessthan 10";

};

}

}

**import** java.util.function.Function;

**public** **class** Test1 {

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

Function<Integer,Double> fun = (f) -> (**double**)f;

**double** d=fun.apply(10);

System.***out***.println("d: "+d);

Function<Integer,Double> fun1 = (f) -> {

**if**(f>10) {

**return** (**double**)f;

}

**else** {

**double** x = (**double**)f;

**return** -x;

}

};

**double** d1=fun1.apply(10);

System.***out***.println("d1: "+d1);

}

}

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

**import** java.util.function.Function;

**public** **class** Check{

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

Function<Integer,String> f = (x)-> {

**if**(x>=0)

**return** "positive";

**else** **return** "negiative";

};

System.***out***.println(f.apply(10));

System.***out***.println(f.apply(-10));

}

}

**First current function will executes later argument function.**

**import** java.util.function.Function;

**public** **class** Check{

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

Function<Integer,Integer> f1 = (x)-> x\*x;

f1= f1.andThen((x)-> x\*4);

System.***out***.println(f1.apply(10));

}

}

**First argument function will executes later current function.**

**import** java.util.function.Function;

**public** **class** Check{

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

Function<Integer,Integer> f1 = (x)-> x\*x;

f1= f1.compose((x)-> x\*4);

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

}

}

**Whatever the value we given the same value will come outside.**

**import** java.util.function.Function;

**public** **class** Check{

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

Function<Integer, Integer> f1 = Function.*identity*();

System.***out***.println(f1);

System.***out***.println(f1.apply(10));

}

}

Example on java.util.function.BiFunction:

Example1:

**import** java.util.function.BiFunction;

**public** **class** Test1 {

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

BiFunction<Integer,Integer,String> b = (n1,n2)-> "the addition is: "+(n1+n2);

String s = b.apply(100, 200);

System.***out***.println("s: "+s);

BiFunction<Integer,Integer,Integer> b1 =

(n1,n2) -> (n1+n2);

**int** i = b1.apply(11,22);

System.***out***.println("i: "+i);

}

}

**import** java.util.function.BiFunction;

**public** **class** Test1 {

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

BiFunction<Integer,Integer,Double> b1 = (n1,n2) ->

{ **if**(n1>10 && n2>10) {

**return** (**double**)(n1+n2);

}

**return** (**double**)(n1-n2);

};

**double** d = b1.apply(50, 60);

System.***out***.println("d: "+d);

**double** d1 = b1.apply(10, 5);

System.***out***.println("d1: "+d1);

}

}

**import** java.util.function.BiFunction;

**public** **class** Test1 {

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

BiFunction<Integer,Integer,Double> b1 = (n1,n2) -> (**double**)(n1+n2);

**double** d = b1.apply(50, 60);

System.***out***.println("d: "+d);

**double** d1 = b1.apply(10, 5);

System.***out***.println("d1: "+d1);

}

}

**import** java.util.function.BiFunction;

**public** **class** Check{

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

BiFunction<String,String,Integer> b1 = (x,y)-> {

**int** a = Integer.*parseInt*(x);

**int** b = Integer.*parseInt*(y);

**return** a\*b;

};

System.***out***.println(b1.apply("4", "9"));

Double finaloutput = b1.andThen((x)->Math.*sqrt*(x)).apply("4", "9");

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

}

}

Program on java.util.function.Consumer:

**import** java.util.function.Consumer;

**public** **class** Test1 {

**public** **static** **void** main(String[] args) { Consumer<Integer> c = (value)-> System.***out***.println(value);

c.accept(10);

Consumer<String> c1 = (value) ->{

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

};

c1.accept("ram");

Consumer<Integer> c2 = (value) ->{

**if**(value >10) {

System.***out***.println("Given value "+value+" is greaterthan 10 ");

}

**else** **if**(value < 10) {

System.***out***.println("Given value "+value+" is lessthan 10 ");

}

**else**

System.***out***.println("Given value "+value+" is equal to 10 ");

};

c2.accept(20);

c2.accept(9);

c2.accept(10);

}

}

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

**import** java.util.ArrayList;

**import** java.util.function.Consumer;

**public** **class** Check{

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

Consumer<Integer> c1= i-> System.***out***.print(i+" ");

Consumer<Integer> c2 = c1.andThen(

i-> System.***out***.println(i\*i));

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

al.add(10);

al.add(20);

al.add(30);

al.add(40);

**for**(Integer ii : al){

c2.accept(ii);

}

}

}

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

**Example on java.util.function.BiConsumer:**

**import** java.util.function.BiConsumer;

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

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

BiConsumer<Integer,Integer> bic = (a,b) -> System.***out***.println(a+b);

bic.accept(100, 200);

BiConsumer<Integer,String> bic1 = (a,b) -> System.***out***.println(a+b);

bic1.accept(100, "good");

BiConsumer<String,String> bic2 = (a,b) -> System.***out***.println(a+b);

bic2.accept("ram", "chandra");

}

}

**import** java.util.function.BiConsumer;

**public** **class** Check{

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

BiConsumer<Integer,Integer> bi1 = (x,y)->{System.***out***.print((x+y)+" ");};

BiConsumer<Integer,Integer> bi2 = (x,y)->{System.***out***.println((x\*y)+" ");};

bi1.andThen(bi2).accept(10, 20);

}

}

**Example on java.util.function.Supplier:**

**import** java.util.Random;

**import** java.util.function.Supplier;

**public** **class** Test1 {

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

Supplier<String> sup = ()-> {

**return** "ram";

};

String s1 = sup.get();

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

Supplier<Integer> sup1 = ()->{

**int**[] a = {10,20,30,40,50};

Random r = **new** Random();

**int** b = r.nextInt(5);

System.***out***.println("b: "+b);

**return** a[b];

};

**int** i1 = sup1.get();

System.***out***.println("i1: "+i1);

**int** i2 = sup1.get();

System.***out***.println("i2: "+i2);

**int** i3 = sup1.get();

System.***out***.println("i3: "+i3);

**int** i4 = sup1.get();

System.***out***.println("i4: "+i4);

**int** i5 = sup1.get();

System.***out***.println("i5: "+i5);

}

}

**Program on BinaryOperator and UnaryOperator:**

**import** java.util.function.BinaryOperator;

**import** java.util.function.UnaryOperator;

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

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

UnaryOperator<Integer> uo = (x) -> x\*10;

**int** i = uo.apply(10);

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

BinaryOperator<String> bo = (x,y) -> x+y;

String s = bo.apply("ram", "chandra");

System.***out***.println("s: "+s);

}

}

Program on maxBy, minBy() of BinaryOperator:

**package** java8newfeatures;

**import** java.util.Comparator;

**import** java.util.function.BinaryOperator;

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

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

BinaryOperator bi = BinaryOperator.*maxBy*(**new** MyComparator());

System.***out***.println(bi.apply(10, 20));

BinaryOperator bi1 = BinaryOperator.*minBy*(**new** MyComparator());

System.***out***.println(bi1.apply(10, 20));

BinaryOperator bi2 = BinaryOperator.*minBy*(**new** MyComparator());

System.***out***.println(bi2.apply(10, 10));

}

}

**class** MyComparator **implements** Comparator{

@Override

**public** **int** compare(Object obj1,Object obj2){

**int** i = (Integer)obj1;

**int** j = (Integer)obj2;

**if**(i<j) **return** -1;

**else** **if**(i>j) **return** +1;

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

}

}

**Program on BooleanSupplier:**

**import** java.util.Scanner;

**import** java.util.function.BooleanSupplier;

**public** **class** Check{

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

BooleanSupplier bs = ()->{

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("enter some value");

**int** value = scan.nextInt();

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

**return** **true**;

**else** **return** **false**;

};

**boolean** result = bs.getAsBoolean();

**if**(result==**true**){

System.***out***.println("given number is even");

}

**else**

System.***out***.println("given number is odd");

}

}

**Program on DoubleBinaryOperator:**

**import** java.util.function.DoubleBinaryOperator;

**public** **class** Check{

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

DoubleBinaryOperator db = (x,y)->(x+y);

**double** d = db.applyAsDouble(10, 20);

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

}

}

**Program on DoubleSupplier:**

**import** java.util.function.DoubleSupplier;

**public** **class** Check{

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

DoubleSupplier ds = ()->{

**return** (2\*3);

};

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

}

}

**Program on DoubleToIntFunction and DoubleToLongFunction :**

**import** java.util.function.DoubleToIntFunction;

**import** java.util.function.DoubleToLongFunction;

**public** **class** Check{

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

DoubleToIntFunction di = (x)-> (**int**)x;

System.***out***.println(di.applyAsInt(12.234));

DoubleToLongFunction dl = (x)-> {**return** (**long**)x;

};

System.***out***.println(dl.applyAsLong(564.321));

}

}

**Functional Interface:**

**An interface which contains only one abstract method is called Functional interface.**

**Interface I{**

**Public abstract void m1();**

**}**

**It may contain more than one default methods.**

**Interface J{**

**Public abstract void m1();**

**Default void m2(){**

**}**

**Default void m3(){**

**}**

**}**

**It may contain more than one static method also.**

**Interface J{**

**Public abstract void m1();**

**static void m2(){**

**}**

**static void m3(){**

**}**

**}**

**We can declare more than one of java.lang.Object class methods.**

@FunctionalInterface

**interface** J{

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

**static** **void** m2(){

}

**static** **void** m3(){

}

**default** **void** m4() {

}

**default** **void** m5() {

}

@Override

**public** String toString();

@Override

**public** **int** hashCode();

}

**It is suitable for developing Functional Programming in java.**

**Example on above all instructions of Function Interface.**

@FunctionalInterface

**interface** J{

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

**static** **void** m2(){

System.***out***.println("static m2 method");

}

**static** **void** m3(){

System.***out***.println("static m3 method");

}

**default** **void** m4() {

System.***out***.println("default m4 method");

}

**default** **void** m5() {

System.***out***.println("default m5 method");

}

@Override

**public** String toString();

@Override

**public** **int** hashCode();

}

**public** **class** Test **implements** J {

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

System.***out***.println("this implementation of abstract method in J");

}

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

J obj = **new** Test();

obj.m1();

J.*m2*();

J.*m3*();

//J.m4();//invalid

//J.m5();//invalid

obj.m4();

obj.m5();

}

}

In the above program in interface ‘J’ there three abstract methods like m1(), toString(), hashCode().  
In our Test class we are giving only implementation for m1(), but not for toString(), hashCode(). In this process we are not getting any error, the reason is by default Test class is the subclass of java.lang. Object. So implementatios for toString() and hashCode() by default coming to Test class from java.lang.Object class.

Example on Functional Interface:

@FunctionalInterface

**interface** J{

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

}

@FunctionalInterface

**interface** K{

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

}

**public** **class** Test **implements** K {

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

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

}

**public** **void** m2() {

System.***out***.println("m2 method");

}

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

K obj = **new** Test();

obj.m2();

}

}

Note 1: In the above program if our interface K implements J then our J is not a functional interface the reason K having two abstract methods one is m1() and second one m2() due to inheritance. It is violating the rules of functional interface.

@FunctionalInterface

**interface** J{

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

}

@FunctionalInterface

**interface** K implements J{

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

}

**Note 2: If we avoid writing @FunctionalInterface annotation on ‘K’ then ‘K’ is becomes a normal interface.**

@FunctionalInterface

**interface** J{

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

}

**interface** K **extends** J{

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

}

**public** **class** Test **implements** K {

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

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

}

**public** **void** m2() {

System.***out***.println("m2 method");

}

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

K obj = **new** Test();

obj.m2();

obj.m1();

}

}

**Is functional interface extends the other normal interface?**

**A: Yes.**

**interface** J{

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

System.***out***.println("defalut method m1() from J interface");

}

}

@FunctionalInterface

**interface** K **extends** J{

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

}

**public** **class** Test **implements** K {

**public** **void** m2() {

System.***out***.println("m2 method");

}

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

K obj = **new** Test();

obj.m2();

obj.m1();

}

}

**Note: Normal interface or non-functional interface (J) should have only default and static methods, if ‘J’ have any abstract method then ‘K’ is not become a functional interface, the reason is due inheritance concept Function interface like ‘K’ will get two methods like m3() and m2(), this is violating the rules of functional interface. Bellow code is invalid.**

**interface** J{

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

System.***out***.println("defalut method m1() from J interface");

}

**public** **abstract** **void** m3();

}

@FunctionalInterface

**interface** K **extends** J{

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

}

**public** **class** Test **implements** K {

**public** **void** m2() {

System.***out***.println("m2 method");

}

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

K obj = **new** Test();

obj.m2();

obj.m1();

}

}

Default Methods in Interface:

In java 8 we can able to write default methods in interface.

These are contains body.

The main advantage of default method is without changing the code in implementation classes we can provide extra functionalities.

Syntax: public default returntype methodname(Params ){

}

**interface** I{

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

}

**class** A **implements** I{

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

System.***out***.println("m1-A");

}

}

**class** B **implements** I{

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

System.***out***.println("m1-B");

}

}

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

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

I obj = **new** A();

obj.m1();

obj = **new** B();

obj.m1();

}

}

**In the above example if we want to add new feature to interface ‘I’, definitely we need to change the code in class ‘A’ and ‘B’. Like bellow**

**interface** I{

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

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

}

**class** A **implements** I{

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

System.***out***.println("m1-A");

}

**public** **void** m2() {

System.***out***.println("m2-A");

}

}

**class** B **implements** I{

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

System.***out***.println("m1-B");

}

**public** **void** m2() {

System.***out***.println("m2-B");

}

}

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

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

I obj = **new** A();

obj.m1();

obj = **new** B();

obj.m1();

}

}

**In the above program we did add m2() method in interface ‘I’, so we did add extra methods in class ‘A’ and ‘B’ also.**

**To avoiding this problem we can do the following things.**

**Without touching existed interface if we want to add new feature then take new interface and extends old interface and implements that interface by A and B.**

**interface** I{

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

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

}

**interface** J **extends** I{

**public** **abstract** **void** m3();

}

**class** A **implements** J{

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

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

**public** **void** m3(){}

}

**class** B **implements** J{

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

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

**public** **void** m3(){}

}

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

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

}

}

In the above scenario also if out classes like A and B extends of J interface still we need to do changes in implementation classes like A,B.

**By using Adapter classes we can solve the above problem like no need change the class like ‘A’ and ‘B’ like bellow:**

**interface** I{

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

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

}

**abstract** **class** AC **implements** I{

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

}

**class** A **extends** AC{

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

System.***out***.println("m1-A");

}

}

**class** B **extends** AC{

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

System.***out***.println("m1-B");

}

}

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

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

I obj = **new** A();

obj.m1();

obj = **new** B();

obj.m1();

}

}

**In above program we taking one adapter class like ‘AC’ and providing empty implementation to m2() and instead of implements ‘I’ by A and B classes we should extends the adapter class like ‘AB’**

**In the above approach also we did some modification on class ‘A’ and ‘B’ like extends of AB (Abstract Class).**

**And we have one more big drawback that is once our classes extends of AB abstract class we can’t extends any other classes, the reason is java doesn’t supports multiple inheritance.**

**Without touching ‘A’ and ‘B’ classes if we want add new feature to interfaces we should use ‘DEFAULT METHOD’ in interface.**

**If we declare any concrete method as default method we should mention ‘default’ keyword in its prototype nothing but method body.**

**Example on default method in interface:**

**interface** I{

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

**default** **public** **void** m2() {

System.***out***.println("this is default method in interface");

}

}

**class** A **implements** I{

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

System.***out***.println("m1-A");

}

}

**class** B **implements** I{

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

System.***out***.println("m1-B");

}

}

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

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

I obj = **new** A();

obj.m1();

obj.m2();

obj = **new** B();

obj.m1();

obj.m2();

}

}

**Default method can participated in overriding also. Like bellow.**

**interface** I{

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

**default** **public** **void** m2() {

System.***out***.println("this is default method in interface");

}

}

**class** A **implements** I{

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

System.***out***.println("m1-A");

}

@Override

**public** **void** m2() {

System.***out***.println("this is override method of interface-A");

}

}

**class** B **implements** I{

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

System.***out***.println("m1-B");

}

@Override

**public** **void** m2() {

System.***out***.println("this is override method of interface-B");

}

}

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

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

I obj = **new** A();

obj.m1();

obj.m2();

obj = **new** B();

obj.m1();

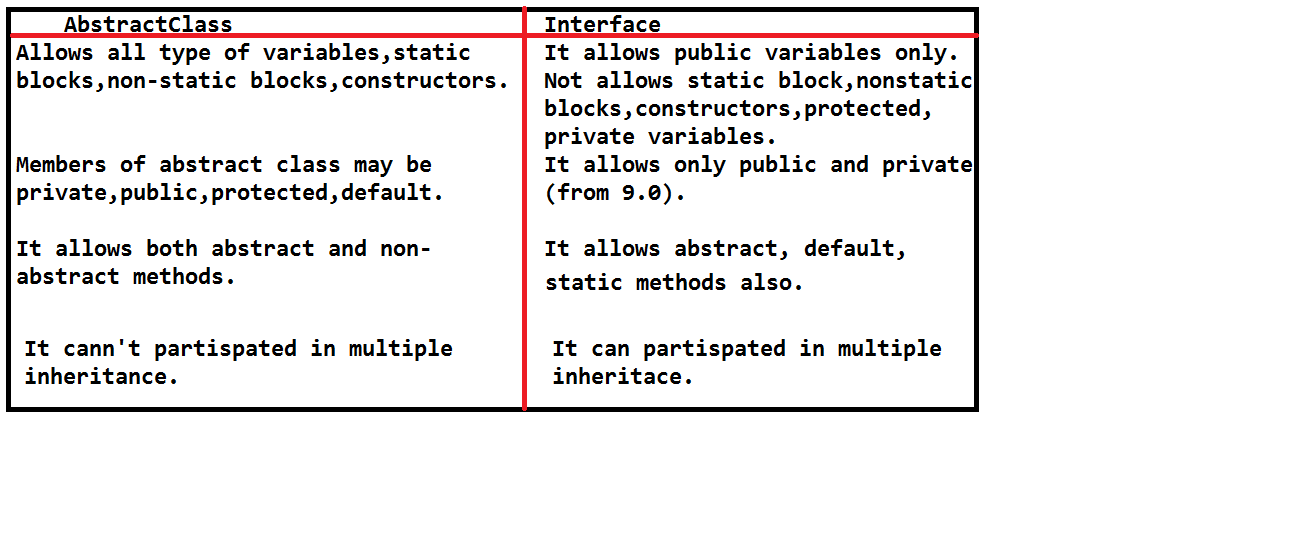
obj.m2();

}

}

**Note: Default methods are allowed only in interface but in class and abstract classes.**

**Difference between abstract class and interfaces:**

****

**Rules of default methods:**

**It participated in method overloading:**

**interface** I{

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

**default** **public** **void** m2() {

System.***out***.println("this is default method in interface");

}

**default** **public** **void** m2(**int** x) {

System.***out***.println("this is default method in interface");

}

}

It can be participated in method overriding.

We can’t override default method as static method.

**interface** I{

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

**default** **public** **void** m2() {

System.***out***.println("this is default method in interface");

}

}

**class** A **implements** I{

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

System.***out***.println("m1-A");

}

//bellow code is invalid.

@Override

**static** **public** **void** m2() {

System.***out***.println("this is override method of interface-A");

}

}

In the above program we will get error.

**Default method should be different from java.lang.Object class methods:**

**interface** I{

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

**default** String toString() {

**return** "ram";

}

}

**Below code will give compile time error.**

**interface** I{

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

System.***out***.println("defalut method of interface-I");

}

}

**interface** J{

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

System.***out***.println("defalut method of interface-J");

}

}

**class** A **implements** I,J{

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

A obj = **new** A();

}

}

**To overcome the above problem we should override default method in our class ‘A’ like bellow.**

**interface** I{

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

System.***out***.println("defalut method of interface-I");

}

}

**interface** J{

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

System.***out***.println("defalut method of interface-J");

}

}

**class** A **implements** I,J{

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

System.***out***.println("defalut method of interface-A");

}

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

A obj = **new** A();

obj.m1();

}

}

We can convert default method as abstract method in abstract class, concrete method we can convert into abstract methods in abstract class. Like bellow.

**interface** I{

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

System.***out***.println("defalut method of interface-I");

}

}

**abstract** **class** AC **implements** I{

/\*@Override

public void m1() {

}\*/

**public** **abstract** **void** m1();//valid

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

**public** **void** m3() {}

}

**abstract** **class** AC1 **extends** AC{

**public** abstract **void** m3();//valid

}

**In the above program AC1 related method will give compile time error.**

**STATIC METHOD IN INTERFACE:**

**If we want to write body method in interface we have concept like default method. To execute interface default method, we should required implementation class of that interface.**

**interface** I{

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

System.***out***.println("default method in interface-I");

}

}

**class** A **implements** I{

}

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

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

I obj = **new** A();

obj.m1();

}

}

**With the support of anonymous inner class we can override default method and we can able to call anonymous inner class override method but not interface default method.**

**interface** I{

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

System.***out***.println("default method in interface-I");

}

}

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

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

I obj1 = **new** I(){

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

System.***out***.println("this is annonymous m1()");

}

};

obj1.m1();

}

}

**Without any implementation class support, if we want execute the body methods of interface, we should go for static methods.**

**interface** I{

**static** **void** m1() {

System.***out***.println("default method in interface-I");

}

}

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

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

I.*m1*();

}

}

**Static methods of interface never override.**

**interface** I{

**static** **void** m1() {

System.***out***.println("default method in interface-I");

}

}

**class** A **implements** I{

@Override **void** m1() {//invalid

}

}

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

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

I obj1 = **new** I(){

@Override

**public** **void** m1() {//invalid

System.***out***.println("this is annonymous m1()");

}

};

}

}

**Streams:**

**A new package like java.util.Stream introduced in java 8.**

**It is the collection of classes, interfaces and enum for doing functional programming.**

**Programming on Collection object copy the data from one ArrayList object to another ArrayList based on some condition.**

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**import** java.util.stream.Collectors;

**public** **class** StreamDemo {

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

ArrayList<Integer> al1 = **new** ArrayList<Integer>();

al1.add(10);

al1.add(20);

al1.add(30);

al1.add(40);

al1.add(50);

al1.add(60);

al1.add(70);

al1.add(80);

ArrayList<Integer> al2 = **new** ArrayList<Integer>();

//without stream api

Iterator<Integer> i = al1.iterator();

**while**(i.hasNext()) {

Integer j = i.next();

**if**(j>=39) {

al2.add(j);

}

}

System.***out***.println(al2);

}

}

**Programming on Collection object copy the data from one ArrayList object to another ArrayList based on some condition.**

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**import** java.util.stream.Collectors;

**public** **class** StreamDemo {

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

ArrayList<Integer> al1 = **new** ArrayList<Integer>();

al1.add(10);

al1.add(20);

al1.add(30);

al1.add(40);

al1.add(50);

al1.add(60);

al1.add(70);

al1.add(80);

ArrayList<Integer> al2 = **new** ArrayList<Integer>();

//without stream api

Iterator<Integer> i = al1.iterator();

**while**(i.hasNext()) {

Integer j = i.next();

**if**(j>=39) {

al2.add(j);

}

}

System.***out***.println(al2);

//with stream api

List<Integer> al3 = al1.stream().filter(v-> v>30).

map(v->v).collect(Collectors.*toList*());

System.***out***.println(al3);

List<Integer> al4 = al1.stream().filter(n->n>50).collect(Collectors.*toList*());

System.***out***.println(al4);

}

}

**Programming on Collection object with user define objects and copy the data from one ArrayList object to another ArrayList based on some condition(age>19).**

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**import** java.util.stream.Collectors;

**class** Students{

**public** **int** sid;

**public** String sname;

**public** **int** sage;

Students(**int** sid,String sname,**int** sage){

**this**.sid = sid;

**this**.sname = sname;

**this**.sage = sage;

}

}

**public** **class** StreamDemo {

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

ArrayList<Students> al1 = **new** ArrayList<Students>();

Students s1 = **new** Students(101,"ram1",25);

Students s2 = **new** Students(102,"ram2",15);

Students s3 = **new** Students(103,"ram3",35);

Students s4 = **new** Students(104,"ram4",5);

Students s5 = **new** Students(105,"ram5",23);

Students s6 = **new** Students(106,"ram6",39);

Students s7 = **new** Students(107,"ram7",18);

Students s8 = **new** Students(108,"ram8",17);

al1.add(s1);

al1.add(s2);

al1.add(s3);

al1.add(s4);

al1.add(s5);

al1.add(s6);

al1.add(s7);

al1.add(s8);

//with stream api

List<Integer> al3 = al1.stream().filter(p -> p.sage > 19)

.map(p -> p.sage).collect(Collectors.*toList*());

System.***out***.println(al3);

List<String> li = al1.stream().filter(p->p.sage>25).map(p->p.sname).

collect(Collectors.*toList*());

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

}

}

ArrayList with Predefine Integer Objects– removeIf();

**import** java.util.ArrayList;

**import** java.util.function.Predicate;

**public** **class** StreamDemo {

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

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

al.add(10);

al.add(20);

al.add(30);

al.add(40);

al.add(50);

al.add(60);

al.add(70);

al.add(80);

System.***out***.println("al: "+al);

Predicate<Integer> p = (p1 ->p1>40);

al.removeIf(p);

System.***out***.println("al: "+al);

}

}

ArrayList with Predefine String Objects– removeIf();

**import** java.util.ArrayList;

**import** java.util.function.Predicate;

**public** **class** StreamDemo {

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

ArrayList<String> al1 = **new** ArrayList<String>();

al1.add("ram");

al1.add("sam");

al1.add("varun");

al1.add("Kiran");

al1.add("baby");

System.***out***.println("al1: "+al1);

Predicate<String> p = (p1->p1.charAt(0)=='r');

al1.removeIf(p);

System.***out***.println("al1: "+al1);

}

}

ArrayList with user define object(Student): removeIf():

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.function.Predicate;

**class** Students{

**public** **int** sid;

**public** String sname;

**public** **int** sage;

Students(**int** sid,String sname,**int** sage){

**this**.sid = sid;

**this**.sname = sname;

**this**.sage = sage;

}

@Override

**public** String toString() {

**return** sid+"..."+sname+"..."+sage;

}

}

**public** **class** StreamDemo {

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

ArrayList<Students> al1 = **new** ArrayList<Students>();

Students s1 = **new** Students(101,"ram1",25);

Students s2 = **new** Students(102,"ram2",15);

Students s3 = **new** Students(103,"ram3",35);

Students s4 = **new** Students(104,"ram4",5);

Students s5 = **new** Students(105,"ram5",23);

Students s6 = **new** Students(106,"ram6",39);

Students s7 = **new** Students(107,"ram7",18);

Students s8 = **new** Students(108,"ram8",17);

al1.add(s1);

al1.add(s2);

al1.add(s3);

al1.add(s4);

al1.add(s5);

al1.add(s6);

al1.add(s7);

al1.add(s8);

System.***out***.println("al1: "+al1);

Predicate<Students> p = (p1->p1.sage>20);

al1.removeIf(p);

Iterator<Students> i = al1.iterator();

**while**(i.hasNext()) {

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

}

}

}

Methods of Predicate:

**import** java.util.ArrayList;

**import** java.util.function.Predicate;

**public** **class** StreamDemo {

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

ArrayList<Integer> al1 = **new** ArrayList<Integer>();

al1.add(6);

al1.add(18);

al1.add(12);

al1.add(15);

al1.add(4);

al1.add(8);

al1.add(24);

al1.add(10);

System.***out***.println("al1: "+al1);

//Predicate<Integer> p = (p1->p1%2==0);

Predicate<Integer> p = (p1->p1%3==0);

al1.removeIf(p);

System.***out***.println("al1: "+al1);

}

}

Example on and() and or() method of predicate:

**import** java.util.ArrayList;

**import** java.util.function.Predicate;

**public** **class** StreamDemo {

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

ArrayList<Integer> al1 = **new** ArrayList<Integer>();

al1.add(6);

al1.add(18);

al1.add(12);

al1.add(15);

al1.add(4);

al1.add(8);

al1.add(24);

al1.add(10);

System.***out***.println("al1: "+al1);

Predicate<Integer> p1 = (p3->p3%2==0);

Predicate<Integer> p2 = (p3->p3%3==0);

System.***out***.print(" : ");

**for**(Integer i : al1) {

/\*if(p1.and(p2).test(i)) {

System.out.print(i+", ");

}\*/

**if**(p1.or(p2).test(i))

System.***out***.print(i+", ");

}

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

System.***out***.println("al1: "+al1);

}

}

Program on predicate to select negative numbers:

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.function.Predicate;

**public** **class** StreamDemo {

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

Predicate<Integer> negative = n -> (n < 0) ;

List<Integer> numbers = Arrays.*asList*(10,20,30,-10,-20,-30);

**long** negativeCount= numbers.stream().filter(negative).count();

System.***out***.println("Count of negative numbers in list numbers = "

+ negativeCount);

}

}

Program on negate() of predicate:

**import** java.util.function.Predicate;

**public** **class** StreamDemo {

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

Predicate<Integer> greaterThanTen = (i) -> i > 10;

// Creating predicate

Predicate<Integer> lowerThanTwenty = (i) -> i < 20;

**boolean** result = greaterThanTen.and(lowerThanTwenty).test(25);

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

// Calling Predicate method

**boolean** result2 = greaterThanTen.and(lowerThanTwenty).negate().test(25);

System.***out***.println(result2);

}

}

Program on isEquals() and test() of predicate:

**import** java.util.Objects;

**import** java.util.function.Predicate;

**public** **class** StreamDemo {

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

**boolean** p = Objects.*equals*("ram", "ram");

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

**boolean** p1 = Objects.*equals*("ram", "ram1");

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

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

Predicate<Integer> p2 = Predicate.*isEqual*(10);

**boolean** b = p2.test(10);

System.***out***.println("b: "+b);

}

}

**Program on forEach() of ArrayList to read the data from Collection object.**

**import** java.util.ArrayList;

**import** java.util.function.Consumer;

**public** **class** StreamDemo {

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

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

al.add(10);

al.add(20);

al.add(30);

al.add(40);

al.add(50);

al.add(60);

al.add(70);

System.***out***.println("al: "+al);

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

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

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

Consumer<Integer> c = (p) ->{System.***out***.println(p);};

al.forEach(c);

}

}

**Program on forEach() of ArrayList by using String objects:**

**import** java.util.ArrayList;

**import** java.util.function.Consumer;

**public** **class** StreamDemo {

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

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

al.add("ram");

al.add("sam");

al.add("kiran");

al.add("varun");

al.add("mahi");

al.add("vani");

System.***out***.println("al: "+al);

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

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

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

Consumer<String> c=(p)->{System.***out***.println(p.toUpperCase());};

al.forEach(c);

}

}

How to create Stream object:

**import** java.util.ArrayList;

**import** java.util.stream.Stream;

**public** **class** StreamDemo {

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

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

al.add(10);

al.add(20);

al.add(30);

al.add(40);

al.add(50);

al.add(60);

al.add(70);

System.***out***.println("al: "+al);

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

Stream<Integer> s = al.stream();

Stream<Integer> s1 = al.stream();

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

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

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

}

}

Program on forEachOrder() of Stream object:

**import** java.util.ArrayList;

**import** java.util.stream.Stream;

**public** **class** StreamDemo {

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

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

al.add(10);

al.add(20);

al.add(40);

al.add(30);

al.add(70);

al.add(60);

al.add(70);

System.***out***.println("al: "+al);

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

Stream<Integer> s = al.stream();

s.forEachOrdered(System.***out***::println);

}

}

Program on filter() of Stream object:

**import** java.util.ArrayList;

**import** java.util.stream.Stream;

**public** **class** StreamDemo {

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

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

al.add("ram");

al.add("sam");

al.add("kiran");

al.add("ramya");

al.add("ravi");

al.add("mahesh");

al.add("vani");

System.***out***.println("al: "+al);

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

Stream<String> s = al.stream();

s.forEachOrdered(System.***out***::println);

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

Stream<String> s1 = al.stream();

Stream<String> s2 = s1.filter((p)->{**return** p.startsWith("r");});

//s2.forEach(System.out::println);

s2.forEachOrdered(System.***out***::println);

}

}

**import** java.util.ArrayList;

**import** java.util.stream.Stream;

**public** **class** StreamDemo {

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

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

al.add(10);

al.add(5);

al.add(15);

al.add(30);

al.add(20);

al.add(40);

al.add(35);

Stream<Integer> s = al.parallelStream();

Stream<Integer> s1 = s.filter((p)->p>20);

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

}

}

Program on map() of stream object

**import** java.util.ArrayList;

**import** java.util.stream.Stream;

**public** **class** StreamDemo {

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

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

al.add("ram");

al.add("sam");

al.add("kiran");

al.add("ramya");

al.add("ravi");

al.add("mahesh");

al.add("vani");

System.***out***.println("al: "+al);

System.***o***

***ut***.println("===================");

Stream<String> s = al.stream();

Stream<Integer> s1 = s.map((v)->{**return** v.length();});

//s1.forEach(System.out::println);

**int** result = s1.mapToInt(Integer::**new**).sum();

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

}

}

Program on IntStream and DoubleStream:

**import** java.util.ArrayList;

**import** java.util.stream.DoubleStream;

**import** java.util.stream.IntStream;

**import** java.util.stream.Stream;

**public** **class** StreamDemo {

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

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

al.add("ram");

al.add("sam");

al.add("kiran");

al.add("ramya");

al.add("ravi");

al.add("mahesh");

al.add("vani");

System.***out***.println("al: "+al);

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

Stream<String> s = al.stream();

Stream<Integer> s1= s.map((v)->{**return** v.length();});

//s1.forEach(System.out::println);

/\*IntStream s2 = s1.mapToInt(Integer::new);

System.out.println(s2.sum());\*/

DoubleStream s3 = s1.mapToDouble(Double::**new**);

System.***out***.println(s3.sum());

}

}

Program on reduce method of Stream object:

**import** java.util.Arrays;

**import** java.util.stream.IntStream;

**public** **class** StreamDemo {

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

**int** a[] = {10,20,30,40,50};

IntStream s = Arrays.*stream*(a);

**int** sum = s.reduce(0,(x,y)->{

**return** (x+y);

}

);

System.***out***.println("sum: "+sum);

}

}

Program max() and min() of Stream objects:

**import** java.util.ArrayList;

**import** java.util.stream.Stream;

**public** **class** StreamDemo {

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

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

al.add(10);

al.add(5);

al.add(15);

al.add(30);

al.add(20);

al.add(40);

al.add(35);

Stream<Integer> s = al.stream();

System.***out***.println(s.max((a,b)->{

**return** a.compareTo(b);

}

));

Stream<Integer> s1 = al.stream();

System.***out***.println(s1.min((a,b)->{

**return** a.compareTo(b);

}

));

}

}

Java.util.StringJoiner:

**public** **class** StreamDemo {

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

String[] s = {"ram","sam","varun"};

String s1 = String.*join*("--", s);

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

}

}

**import** java.util.StringJoiner;

**public** **class** StreamDemo {

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

StringBuilder s1 = **new** StringBuilder();

s1.append("ram");

s1.append("kiran");

s1.append("varun");

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

StringJoiner sj = **new** StringJoiner("\_\_");

sj.add("ram");

sj.add("kiran");

sj.add("varun");

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

}

}

**import** java.util.StringJoiner;

**public** **class** StreamDemo {

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

StringJoiner sj1 = **new** StringJoiner("--", "[", "]");

sj1.add("ram");

sj1.add("kiran");

sj1.add("varun");

System.***out***.println(sj1);

StringJoiner sj2 = **new** StringJoiner(",");

sj2.add("friends");

sj2.add("relatives");

System.***out***.println(sj2);

sj1.merge(sj2);

System.***out***.println(sj1);

**StringJoiner st3 = new StringJoiner(",");**

**System.*out*.println(st3.length());**

**st3.add("kiran");**

**System.*out*.println(st3.length());**

}

}

**import** java.util.ArrayList;

**import** java.util.StringJoiner;

**public** **class** StreamDemo {

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

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

al.add("ram");

al.add("kiran");

al.add("varun");

System.***out***.println("al: "+al);

StringJoiner sj = **new** StringJoiner("-");

**for**(String s1 : al) {

sj.add(s1);

}

System.***out***.println("sj: "+sj);

String s2 = sj.toString();

System.***out***.println("s2: "+s2);

}

}

Optional Class in java:

**public** **class** Check{

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

String[] s = **new** String[10];

s[0]="ram";

System.***out***.println(s[0].toUpperCase());

System.***out***.println(s[9].toUpperCase());

}

}

In the above scenario we will get one output is RAM and later we will get NullPointerException.

To overcome this problem we have alternative like Optional class.

**import** java.util.Optional;

**public** **class** StreamDemo {

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

String[] s = {"ram","kiran","varun","mahi","hi"};

System.***out***.println(s[4].toUpperCase());

Optional<String> o = Optional.*ofNullable*(s[3]);

**if**(o.isPresent()) {

System.***out***.println(s[3].toUpperCase());

}

**else** {

System.***out***.println("there is no value in that index");

}

}

}

In the above program we have s[3] value like mahi it will converting into MAHI.

**import** java.util.Optional;

**public** **class** StreamDemo {

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

String[] s = **new** String[15];

Optional<String> o = Optional.*ofNullable*(s[3]);

**if**(o.isPresent()) {

System.***out***.println(s[3].toUpperCase());

}

**else** {

System.***out***.println("there is no value in that index");

}

}

}

In the above scenario there is no value in that index position but still we are not getting exception simply it will showing like following message on console.

There is no value in that index.

**import** java.util.Optional;

**public** **class** StreamDemo {

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

String[] s = **new** String[15];

Optional<String> o = Optional.*ofNullable*(s[3]);

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

s[3]="ram chandra rao";

Optional<String> o1 = Optional.*ofNullable*(s[3]);

o1.ifPresent(System.***out***::println);

System.***out***.println(o1.get());

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

}

}

**import** java.util.Optional;

**public** **class** StreamDemo {

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

String[] str = **new** String[10];

str[5] = "ram";

Optional<String> empty = Optional.*empty*();

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

Optional<String> value = Optional.*of*(str[5]);

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

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

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

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

System.***out***.println(value.*ofNullable*(str[5]));

System.***out***.println(empty.orElse("value is not present"));

System.***out***.println(value.orElse("value is not present"));

}

}

**import** java.util.Optional;

**import** java.util.stream.Stream;

**public** **class** StreamDemo {

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

Stream <Integer> s = Stream.*of*(10,20,30,40,50);

// s.forEach(System.out::println);

Optional<Integer> s1 = s.findFirst();

**if**(s1.isPresent()) {

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

}

}

}

**import** java.util.Optional;

**public** **class** Check{

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

String[] s = **new** String[10];

s[0]="ram";

Optional o = Optional.*empty*();

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

//System.out.println(o.get()); //NoSuchElementException : no value present

//to overcome above problem first we have to check whether we Optional call have value or not

**if**(o.isPresent()){

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

}

**else**{

System.***out***.println("value is not present");

}

Optional<String> o1 = Optional.*of*(s[0]);

System.***out***.println(o1);

**if**(o1.isPresent()){

System.***out***.println(o1.get()+"..."+o1.get().hashCode());

String s1 = (String)o1.get();

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

}

**else**{

System.***out***.println("value is not present");

}

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

o1.ifPresent((x)->{System.***out***.println(x);});

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

Optional<String> o2 = Optional.*ofNullable*(s[0]);

System.***out***.println(o2.get());

Optional<String> o3 = Optional.*ofNullable*(s[1]);

System.***out***.println(o3.isPresent());

System.***out***.println(o3.orElse("value is not present in 1 index position"));

System.***out***.println(o2.orElse("value is not present in 1 index position"));

Optional<String> o4 = o2.filter((x)->{

**boolean** b = x.equals("RAM");

**return** b;

});

System.***out***.println(o4+"..."+o4.isPresent());

Optional o5 = o2.map((x)->x.length());

System.***out***.println(o5);

}

}

Java Nashorn

It is java script engine. We can call and execute the java script code directly from our command prompt by using following command like “jjs”.

Following steps are useful to work with java nashorn.

1. Take one notepad.
2. Type following code in that notepad.

var message = function(){

print("this is new feature.....");

};

message();

1. Save above file with name like message.js.
2. Open command prompt and type bellow command

jjs message.js

We can able to execute the above message.js file by using java program also.

import javax.script.\*;

import java.io.\*;

public class Demo{

public static void main(String[] args) throws Exception{

ScriptEngine ee = new ScriptEngineManager().getEngineByName("Nashorn");

ee.eval(new FileReader( "C:\\Users\\Ramchandar\\ Desktop\\message.js"));

}

}

Javac Demo.java

Java Demo

Note: we should write in the above program either “Nashorn” or “nashorn”, otherwise we will get java.lang.NullPointerException.

ParallelArraySoring:

**import** java.util.Arrays;

**public** **class** ParallelArraySorting {

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

**int**[] arr = {5,8,1,0,6,9};

**for** (**int** i : arr) {

System.***out***.print(i+" ");

}

Arrays.*parallelSort*(arr);

System.***out***.println("\nArray elements after sorting");

**for** (**int** i : arr) {

System.***out***.print(i+" ");

}

}

}

ParallelArraySoring by using specific index positions:

**import** java.util.Arrays;

**public** **class** ParallelArraySorting {

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

**int**[] arr = {5,8,1,0,6,9,50,-3};

**for** (**int** i : arr) {

System.***out***.print(i+" ");

}

Arrays.*parallelSort*(arr,0,4);

System.***out***.println("\nArray elements after sorting");

**for** (**int** i : arr) {

System.***out***.print(i+" ");

}

}

}

**Working with Date and Time api:**

**Program on LocalDate class:**

**import** java.time.LocalDate;

**public** **class** DateDemo {

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

//current date

LocalDate ld = LocalDate.*now*();

System.***out***.println("ld: "+ld);

String sld = ld.toString();

System.***out***.println("sld: "+sld);

//our own date

//LocalDate ld1 = LocalDate.of(1987, 12, 16);

LocalDate ld1 = LocalDate.*of*(2000, 12, 16);

System.***out***.println("ld1: "+ld1);

**boolean** b = ld1.isLeapYear();

**if**(b) {

System.***out***.println("given year is leap year");

}

**else** {

System.***out***.println("given year is non-leap year");

}

}

}

**Manipulating date and time by using plus and minus methods of LocalDate:**

**import** java.time.LocalDate;

**public** **class** DateDemo {

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

//current date

LocalDate ld = LocalDate.*now*();

System.***out***.println("ld: "+ld);

LocalDate ld1 = LocalDate.***MAX***;

System.***out***.println("ld1: "+ld1);

LocalDate ld2 = LocalDate.***MIN***;

System.***out***.println("ld2: "+ld2);

LocalDate ld3 = ld.plusDays(1);

System.***out***.println("ld3: "+ld3);

LocalDate ld4 = ld.plusMonths(2);

System.***out***.println("ld4: "+ld4);

LocalDate ld5 = ld.plusWeeks(2);

System.***out***.println("ld5: "+ld5);

LocalDate ld6 = ld.plusYears(2);

System.***out***.println("ld6: "+ld6);

LocalDate ld7 = ld.minusDays(2);

System.***out***.println("ld7: "+ld7);

LocalDate ld8 = ld.minusYears(2);

System.***out***.println("ld8: "+ld8);

LocalDate ld9 = ld.minusMonths(2);

System.***out***.println("ld9: "+ld9);

LocalDate ld10 = ld.minusWeeks(2);

System.***out***.println("ld10: "+ld10);

}

}

**Program on to get individual years, months, days, weak from LocalDate:**

**import** java.time.DayOfWeek;

**import** java.time.LocalDate;

**import** java.time.Month;

**public** **class** DateDemo {

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

//current date

LocalDate ld = LocalDate.*now*();

System.***out***.println("ld: "+ld);

**int** year = ld.getYear();

System.***out***.println("year: "+year);//2018

**int** month = ld.getMonthValue();

System.***out***.println("month: "+month);//5

Month m = ld.getMonth();

System.***out***.println("M: "+m);//MAY

**int** day = ld.getDayOfMonth();

System.***out***.println("day: "+day);//11

**int** dayYear = ld.getDayOfYear();

System.***out***.println("dayYear: "+dayYear);//131

DayOfWeek dayWeek = ld.getDayOfWeek();

System.***out***.println("dayWeek: "+dayWeek);//FRIDAY

//String to LocalDate

//String s = "1987-13-16";//month should be range

String s = "1987-12-16";

LocalDate ld1 = LocalDate.*parse*(s);

System.***out***.println("ld1: "+ld1);

//String s1 = "1987-12-32";//day should be range

String s1 = "1987-12-31";

LocalDate ld2 = LocalDate.*parse*(s1);

System.***out***.println("ld2: "+ld2);

}

}

**Program on LocalDateTime:**

**import** java.time.DayOfWeek;

**import** java.time.LocalDateTime;

**import** java.time.Month;

**public** **class** DateDemo {

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

LocalDateTime ldt = LocalDateTime.*now*();

System.***out***.println("ldt: "+ldt);

**int** hour = ldt.getHour();

**int** min = ldt.getMinute();

**int** sec = ldt.getSecond();

**int** nsec = ldt.getNano();

System.***out***.println("hour: "+hour);

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

System.***out***.println("sec: "+sec);

System.***out***.println("nsec: "+nsec);

**int** day = ldt.getDayOfMonth();

System.***out***.println("day: "+day);

**int** day1 = ldt.getDayOfYear();

System.***out***.println("day1: "+day1);

DayOfWeek day2 = ldt.getDayOfWeek();

System.***out***.println("day2: "+day2);

Month month = ldt.getMonth();

System.***out***.println("month: "+month);

**int** mvalue = ldt.getMonthValue();

System.***out***.println("mvalue: "+mvalue);

**int** year = ldt.getYear();

System.***out***.println("year: "+year);

}

}

**Program on LocalTime:**

**import** java.time.LocalTime;

**public** **class** DateDemo {

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

LocalTime lt = LocalTime.*now*();

System.***out***.println("lt: "+lt);

**int** h = lt.getHour();

**int** m = lt.getMinute();

**int** s = lt.getSecond();

**int** ns= lt.getNano();

System.***out***.println("h: "+h);

System.***out***.println("m: "+m);

System.***out***.println("s: "+s);

System.***out***.println("ns: "+ns);

}

}

Program On Calendar:

**import** java.util.Calendar;

**import** java.util.Date;

**public** **class** DateDemo {

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

Calendar c = Calendar.*getInstance*();

System.***out***.println("c: "+c);

Date d = c.getTime();

System.***out***.println("d: "+d);

/\*c.add(Calendar.MONTH, 1);

Date d1 = c.getTime();

System.out.println("d1: "+d1);\*/

/\*c.add(Calendar.DATE,1);

Date d2 = c.getTime();

System.out.println("d2: "+d2);\*/

c.add(Calendar.***YEAR***, 2);

Date d3 = c.getTime();

System.***out***.println("d3: "+d3);

}

}

**import** java.util.Calendar;

**import** java.util.Date;

**public** **class** DateDemo {

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

Calendar c = Calendar.*getInstance*();

System.***out***.println("c: "+c);

System.***out***.println(c.get(Calendar.***YEAR***));

System.***out***.println(c.get(Calendar.***MONTH***));

System.***out***.println(c.get(Calendar.***DATE***));

System.***out***.println(c.get(Calendar.***DAY\_OF\_MONTH***));

System.***out***.println(c.get(Calendar.***DAY\_OF\_WEEK***));

System.***out***.println(c.get(Calendar.***DAY\_OF\_YEAR***));

}

}

**import** java.util.Calendar;

**public** **class** DateDemo {

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

Calendar c = Calendar.*getInstance*();

System.***out***.println("c: "+c);

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

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

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

}

}

**import** java.util.Calendar;

**public** **class** DateDemo {

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

Calendar c = Calendar.*getInstance*();

System.***out***.println("c: "+c);

System.***out***.println(c.getMaximum(Calendar.***DAY\_OF\_MONTH***));

System.***out***.println(c.getMaximum(Calendar.***DAY\_OF\_WEEK***));

System.***out***.println(c.getMaximum(Calendar.***DAY\_OF\_YEAR***));

}

}

**import** java.util.Calendar;

**public** **class** DateDemo {

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

Calendar c = Calendar.*getInstance*();

System.***out***.println("c: "+c);

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

System.***out***.println(c.getMinimum(Calendar.***DAY\_OF\_WEEK***));

System.***out***.println(c.getMinimum(Calendar.***DAY\_OF\_YEAR***));

System.***out***.println(c.getMinimum(Calendar.***DAY\_OF\_MONTH***));

}

}

**Program Timestamp:**

**import** java.sql.Time;

**import** java.sql.Timestamp;

**import** java.text.SimpleDateFormat;

**import** java.util.Date;

**public** **class** DateDemo {

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

**long** time = System.*currentTimeMillis*();

Timestamp ts = **new** Timestamp(time);

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

String timeStamp = **new** SimpleDateFormat("yyyy.MM.dd.HH.mm.ss").format(**new** Date());

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

String timeStamp1 = **new** SimpleDateFormat("yy/MMM/dd").format(**new** Date());

System.***out***.println(timeStamp1);

Time ts1 = **new** ~~Time~~(12, 12, 12);

System.***out***.println(ts1);

System.***out***.println(ts1.~~getHours~~());

System.***out***.println(ts1.~~getMinutes~~());

System.***out***.println(ts1.~~getSeconds~~());

}

}

**import java.util.Scanner;**

**import java.util.function.BooleanSupplier;**

**import java.util.function.DoubleBinaryOperator;**

**import java.util.function.DoubleConsumer;**

**public class JdbcDemo {**

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

**/\*Predicate<Integer> pi= (Integer n) ->{**

**if(n%2==0){**

**return true;**

**}**

**else{**

**return false;**

**}**

**};**

**System.out.println(pi.test(15));**

**System.out.println(pi.test(10));**

**\*/**

**/\*Predicate<Integer> pi= (n) ->{**

**if(n%2==0){**

**return true;**

**}**

**else{**

**return false;**

**}**

**};**

**System.out.println(pi.test(15));**

**System.out.println(pi.test(10));**

**\*/**

**/\*Predicate<Integer> pi= n ->{**

**if(n%2==0){**

**return true;**

**}**

**else{**

**return false;**

**}**

**};**

**System.out.println(pi.test(15));**

**System.out.println(pi.test(10));**

**\*/**

**/\*Function<Integer, String> fun = (Integer i) ->{**

**if(i%2==0){**

**return "Given Number "+i+" is even number";**

**}**

**else{**

**return "Given Number "+i+" is even odd";**

**}**

**};**

**System.out.println(fun.apply(10));**

**System.out.println(fun.apply(15));\*/**

**/\*Function<Integer, String> fun = (i) ->{**

**if(i%2==0){**

**return "Given Number "+i+" is even number";**

**}**

**else{**

**return "Given Number "+i+" is even odd";**

**}**

**};**

**System.out.println(fun.apply(10));**

**System.out.println(fun.apply(15));\*/**

**/\*Function<Integer, String> fun = i ->{**

**if(i%2==0){**

**return "Given Number "+i+" is even number";**

**}**

**else{**

**return "Given Number "+i+" is even odd";**

**}**

**};**

**System.out.println(fun.apply(10));**

**System.out.println(fun.apply(15));\*/**

**/\*** Function<String,String> fun =

(String s) -> {

//return s.toUpperCase();

return

new StringBuffer(s).reverse().toString().toUpperCase();

};

for(int i=1;i<=5;i++){

System.*out*.println("enter string");

String s = scan.next();

System.*out*.println(s+"..."+fun.apply(s));

}

**\*/**

**/\*Consumer<Integer> con = (Integer i)-> {**

**if(i>=0){**

**System.out.println("given number is positive");**

**}**

**else{**

**System.out.println("given number is negative");**

**}**

**};**

**con.accept(100);**

**con.accept(-100);\*/**

**/\* Consumer<Integer> con = i-> {**

**if(i>=0){**

**System.out.println("given number is positive");**

**}**

**else{**

**System.out.println("given number is negative");**

**}**

**};**

**con.accept(-100);**

**con.accept(+100);\*/**

**/\*Consumer<Integer> con =i-> {**

**if(i>=0){**

**System.out.println("given number is positive");**

**}**

**else{**

**System.out.println("given number is negative");**

**}**

**};**

**con.accept(-100);**

**con.accept(-100);\*/**

**/\* //int sid=100;**

**int sid = -100;**

**Supplier<String> sup = ()->{**

**if(sid>=0){**

**return "given number is positive";**

**}**

**else{**

**return "given number is negative";**

**}**

**};**

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

**\*/**

**/\*int sid = -100;**

**Supplier<String> sup = ()->{**

**int id=100;**

**if(id>=0){**

**return "given number is positive";**

**}**

**else{**

**return "given number is negative";**

**}**

**};**

**System.out.println(sup.get());\*/**

**/\*BiFunction<Integer, Integer, String> bifun =**

**(Integer i, Integer j)->{**

**int result = i+j;**

**if(result >=0){**

**return result+" is positive: ";**

**}**

**else{**

**return result+" is negative: ";**

**}**

**};**

**System.out.println(bifun.apply(100, -200));**

**System.out.println(bifun.apply(-100, 200));\*/**

**/\*BiFunction<Integer, Integer, String> bifun =**

**(i, j)->{**

**int result = i+j;**

**if(result >=0){**

**return result+" is positive: ";**

**}**

**else{**

**return result+" is negative: ";**

**}**

**};**

**System.out.println(bifun.apply(100, -200));**

**System.out.println(bifun.apply(-100, 200));\*/**

**/\*BiFunction<Integer, Integer, String> bifun =**

**i,j->{ //without parenthesis we can write two variables**

**int result = i+j;**

**if(result >=0){**

**return result+" is positive: ";**

**}**

**else{**

**return result+" is negative: ";**

**}**

**};**

**System.out.println(bifun.apply(100, -200));**

**System.out.println(bifun.apply(-100, 200));\*/**

**/\*BiConsumer<Integer,Integer> bicon = (Integer i,Integer j)->{**

**int result=(i+j);**

**if(result>=0){**

**System.out.println(result+". is positive");**

**}**

**else{**

**System.out.println(result+". is negative");**

**}**

**};**

**bicon.accept(100, -200);**

**bicon.accept(-100, +200);**

**\*/**

**/\* BiPredicate<Integer ,Integer> bip = (Integer i,Integer j)->{**

**int result=i+j;**

**if(result>= 0){**

**return true;**

**}**

**else{**

**return false;**

**}**

**};**

**System.out.println(bip.test(-100, -200));**

**System.out.println(bip.test(100, 200));**

**\*/**

**/\*BinaryOperator<String> biop =**

**(String i,String j)->{**

**String s3 = (i+j);**

**return s3;**

**};**

**String s1 = biop.apply("ram", "chandra");**

**String s2 = biop.apply("core", "java");**

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

**System.out.println("s2: "+s2);**

**\*/**

**/\* UnaryOperator<Integer> uo = (Integer i)->{**

**int result = (int)Math.sqrt(i);**

**return result;**

**};**

**System.out.println("The square of the given number is:"+uo.apply(1225));**

**\*/**

**/\*BooleanSupplier bs = ()->{**

**System.out.println("Enter a number");**

**int j = new Scanner(System.in).nextInt();**

**if(j%2==0){**

**return true;**

**}**

**else{**

**return false;**

**}**

**};**

**boolean b = bs.getAsBoolean();**

**if(b){**

**System.out.println("given number is even number");**

**}**

**else{**

**System.out.println("given number is odd number");**

**}\*/**

**/\*DoubleBinaryOperator dbo = (i,j)->{**

**double d = Math.sqrt(i)+Math.sqrt(j);**

**return d;**

**};**

**double result=dbo.applyAsDouble(100, 25);**

**System.out.println("The result is: "+result);\*/**

**DoubleConsumer dc = (double d)->{**

**double d1=Math.*sqrt*(d);**

**System.*out*.println("d1: "+d1);**

**};**

**dc.accept(100.00);**

**}**

**}**

**import java.util.ArrayList;**

**import java.util.List;**

**import java.util.stream.Collectors;**

**import java.util.stream.Stream;**

**public class JdbcDemo {**

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

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

**l.add(0);**

**l.add(10);**

**l.add(15);**

**l.add(20);**

**l.add(25);**

**l.add(30);**

**System.*out*.println("l: "+l);**

**/\*List<Integer> l2 = new ArrayList<Integer>();**

**for(Integer l1 : l ){**

**if(l1%2==0){**

**l2.add(l1);**

**}**

**}**

**System.out.println("l2: "+l2);\*/**

**/\*Stream<Integer> s = l2.stream();**

**Stream<Integer> s1 = s.filter(**

**(Integer i)->{**

**if(i%2==0){**

**return true;**

**}**

**else return false;**

**}**

**);**

**List<Integer> l1 = s1.collect(Collectors.toList());**

**System.out.println("l1: "+l1);**

**i \*/**

**/\*Stream<Integer> s = l2.stream();**

**Stream<Integer> s1 = s.filter(i->i%2==0);**

**List<Integer> l1 = s1.collect(Collectors.toList());**

**System.out.println("l1: "+l1);\*/**

**/\*List<Integer> l1 = l2.stream().filter(i->i%2==0).**

**collect(Collectors.toList());**

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

**/\*Stream<Integer> ss = l.stream();**

**Stream<Integer> ss1 = ss.map(**

**(i)->{**

**return i\*2;**

**} );**

**List<Integer> ss2 = ss1.collect(Collectors.toList());**

**System.out.println(ss2);\*/**

**/\*List<Integer> ss3 = l.stream().map(i->i\*2).**

**collect(Collectors.toList());**

**System.out.println(ss3);\*/**

**List<Integer> ss4 = l.stream().**

**filter(i->i%2==0).**

**map(i->i\*2).collect(Collectors.*toList*());**

**System.*out*.println(ss4);**

**}**

**}**

**import java.util.ArrayList;**

**import java.util.List;**

**import java.util.stream.Collectors;**

**import java.util.stream.Stream;**

**class Book{**

**String bookname;**

**int price;**

**Book(String bookname,int price){**

**this.bookname=bookname;**

**this.price = price;**

**}**

**public String toString(){**

**return bookname+"..."+price;**

**}**

**}**

**public class JdbcDemo {**

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

**Book b1 = new Book("cj",500);**

**Book b2 = new Book("aj",1500);**

**Book b3 = new Book("oj",2500);**

**Book b4 = new Book("pj",3500);**

**Book b5 = new Book("hj",4500);**

**List<Book> lb1 = new ArrayList<Book>();**

**lb1.add(b1);**

**lb1.add(b2);**

**lb1.add(b3);**

**lb1.add(b4);**

**lb1.add(b5);**

**System.*out*.println("lb1: "+lb1);**

**List<Integer> lb3 = new ArrayList<Integer>();**

**for(Book lb2: lb1){**

**//System.out.println(lb2.bookname+"..."+lb2.price);**

**if(lb2.price<3000){**

**lb3.add(lb2.price);**

**}**

**}**

**System.*out*.println(lb3);**

**/\*Stream<Book> sb = lb1.stream();**

**Stream<Book> sb1=sb.filter(b->b.price<=3000);**

**List<Book> sb2 = sb1.collect(Collectors.toList());**

**System.out.println(sb2);**

**\*/**

**Stream<Book> sb = lb1.stream();**

**Stream<Book> sb1=sb.filter(b->b.price<=3000);**

**Stream<Integer> sb2 = sb1.map(b->b.price);**

**List<Integer> sb3 = sb2.collect(Collectors.*toList*());**

**System.*out*.println(sb3);**

**}**

**}**

**List<Integer> sb4 = lb1.stream().filter(b->b.price<=3000)**

**.map(b->b.price).collect(Collectors.*toList*());**

**System.*out*.println(sb4);**

**lb1.stream().filter(b->b.price<=3000).**

**forEach(b->System.*out*.println(b.price));**

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

**Stream<Integer> a1 = Stream.*iterate*(1, n->n+1);**

**System.*out*.println(a1.map(i->i).filter(i->i%2==0).**

**limit(5).collect(Collectors.*toList*()));**

**/\*Find out even number in a ArrayList collection**

**object and to one more collection object by using filter(-)\*/**

**import java.util.ArrayList;**

**import java.util.List;**

**import java.util.stream.Collectors;**

**public class Test{**

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

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

**al.add(10);**

**al.add(0);**

**al.add(20);**

**al.add(15);**

**al.add(25);**

**al.add(30);**

**System.*out*.println("al: "+al);**

**List<Integer> al1 = new ArrayList<Integer>();**

**//with out stream**

**for(Integer i1: al){**

**if(i1%2==0){**

**al1.add(i1);**

**}**

**}**

**System.*out*.println("al1: "+al1);**

**//with stream**

**List<Integer> al2 = al.stream().**

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

**System.*out*.println("al2: "+al2);**

**}**

**}**

**/\*Find out square of every number in a ArrayList collection**

**object and copy into one more collection object by using map(-)\*/**

**import java.util.ArrayList;**

**import java.util.List;**

**import java.util.stream.Collectors;**

**public class Test{**

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

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

**al.add(10);**

**al.add(0);**

**al.add(20);**

**al.add(15);**

**al.add(25);**

**al.add(30);**

**System.*out*.println("al: "+al);**

**List<Integer> al1 = new ArrayList<Integer>();**

**//with out stream**

**for(Integer i1: al){**

**i1=i1\*i1;**

**al1.add(i1);**

**}**

**System.*out*.println("al1: "+al1);**

**//with stream**

**List<Integer> al2 = al.stream().**

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

**System.*out*.println("al2: "+al2);**

**}**

**}**

**/\*Find out even number in a ArrayList collection**

**object and find out square of a number**

**and copy into one more collection object by using filter(-) and map(-)\*/**

**import java.util.ArrayList;**

**import java.util.List;**

**import java.util.stream.Collectors;**

**public class Test{**

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

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

**al.add(10);**

**al.add(0);**

**al.add(20);**

**al.add(15);**

**al.add(25);**

**al.add(30);**

**System.*out*.println("al: "+al);**

**List<Integer> al1 = new ArrayList<Integer>();**

**//with out stream**

**for(Integer i1: al){**

**if(i1%2==0){**

**i1=i1\*i1;**

**al1.add(i1);**

**}**

**}**

**System.*out*.println("al1: "+al1);**

**//with stream**

**List<Integer> al2 = al.stream().**

**filter(i->i%2==0).map(i->i\*i).**

**collect(Collectors.*toList*());**

**System.*out*.println("al2: "+al2);**

**}**

**}**

**/\* Find out number of elements which are less than given number**

**by using filter(-) and count() methods.**

**\*/**

**import java.util.ArrayList;**

**import java.util.List;**

**import java.util.stream.Collectors;**

**public class Test{**

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

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

**al.add(10);**

**al.add(0);**

**al.add(20);**

**al.add(15);**

**al.add(60);**

**al.add(30);**

**al.add(40);**

**al.add(45);**

**al.add(50);**

**al.add(70);**

**al.add(75);**

**al.add(89);**

**System.*out*.println("al: "+al);**

**//with out stream**

**int count=0;**

**for(Integer i1: al){**

**if(i1<50){**

**++count;**

**}**

**}**

**System.*out*.println("count: "+count);**

**//with stream**

**int count1 = (int) al.stream().filter(i->i<50).count();**

**System.*out*.println("count1: "+count1);**

**}**

**}**

**/\* sorting the elements of ArrayList object by using sorted()**

**\* and copied into another ArrayList**

**\*/**

**import java.util.ArrayList;**

**import java.util.List;**

**import java.util.stream.Collectors;**

**public class Test{**

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

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

**al.add(10);**

**al.add(0);**

**al.add(20);**

**al.add(15);**

**al.add(60);**

**al.add(30);**

**al.add(40);**

**al.add(45);**

**al.add(50);**

**al.add(70);**

**al.add(75);**

**al.add(89);**

**System.*out*.println("al : "+al);**

**//with stream**

**List<Integer> al1 = al.stream().**

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

**System.*out*.println("al1: "+al1);**

**}**

**}**

**/\* user define sorting on the elements of ArrayList object by using sorted()**

**\* and copied into another ArrayList.**

**\*/**

**import java.util.ArrayList;**

**import java.util.List;**

**import java.util.stream.Collectors;**

**public class Test{**

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

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

**al.add(10);**

**al.add(0);**

**al.add(15);**

**al.add(60);**

**al.add(30);**

**al.add(40);**

**al.add(45);**

**al.add(50);**

**al.add(70);**

**al.add(75);**

**al.add(89);**

**System.*out*.println("al : "+al);**

**//with stream**

**List<Integer> al1 = al.stream().**

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

**System.*out*.println("al1: "+al1);**

**List<Integer> al2 = al.stream().**

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

**System.*out*.println("al2: "+al2);**

**List<Integer> al3 = al.stream().**

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

**System.*out*.println("al3: "+al3);**

**List<Integer> al4 = al.stream().**

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

**System.*out*.println("al4: "+al4);**

**}**

**}**

**/\* user define sorting on the elements of ArrayList object by using sorted()**

**\* and copied into another ArrayList.**

**\*/**

**import java.util.ArrayList;**

**import java.util.Comparator;**

**import java.util.List;**

**import java.util.stream.Collectors;**

**public class Test{**

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

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

**al.add("ram");**

**al.add("rams123");**

**al.add("rams");**

**al.add("ravi");**

**al.add("raj");**

**al.add("reena");**

**System.*out*.println("al : "+al);**

**//with stream**

**List<String> al1 = al.stream().sorted((i1,i2)->i1.compareTo(i2)).**

**collect(Collectors.*toList*());**

**System.*out*.println("al1: "+al1);**

**List<String> al2 = al.stream().sorted((i1,i2)->-i1.compareTo(i2)).**

**collect(Collectors.*toList*());**

**System.*out*.println("al2: "+al2);**

**List<String> al3 = al.stream().sorted((i1,i2)->i2.compareTo(i1)).**

**collect(Collectors.*toList*());**

**System.*out*.println("al3: "+al3);**

**Comparator<String> c = (i1,i2) ->{**

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

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

**if(i<j) return +1;**

**else if(i>j) return -1;**

**else return i1.compareTo(i2);**

**};**

**List<String> al4 = al.stream().sorted(c).**

**collect(Collectors.*toList*());**

**System.*out*.println("al4: "+al4);**

**}**

**}**

**/\* Find out min and max elements of ArrayList by using min(-) and max(-)**

**\* methods**

**\*/**

**import java.util.ArrayList;**

**import java.util.List;**

**public class Test{**

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

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

**al.add(60);**

**al.add(30);**

**al.add(50);**

**al.add(20);**

**al.add(70);**

**al.add(40);**

**al.add(80);**

**System.*out*.println("al : "+al);**

**//with stream**

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

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

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

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

**int min1 = al.stream().min((i1,i2)->-i1.compareTo(i2)).get();**

**System.*out*.println("min1: "+min1);**

**int max1 = al.stream().max((i1,i2)->-i1.compareTo(i2)).get();**

**System.*out*.println("max1: "+max1);**

**}**

**}**

**/\* Iterate elememts of ArrayList by using forEach(-)**

**\*/**

**import java.util.ArrayList;**

**import java.util.List;**

**import java.util.function.Consumer;**

**public class Test{**

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

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

**al.add(60);**

**al.add(30);**

**al.add(50);**

**al.add(20);**

**al.add(70);**

**al.add(40);**

**al.add(80);**

**System.*out*.println("al : "+al);**

**//without stream**

**for(Integer i1: al)**

**System.*out*.println(i1);**

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

**//with stream**

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

**Consumer<Integer> c = (i)->{**

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

**};**

**al.stream().forEach(c);**

**}**

**}**

**/\* converting ArrayList to array**

**\*/**

**import java.util.ArrayList;**

**import java.util.Arrays;**

**import java.util.List;**

**import java.util.function.Consumer;**

**public class Test{**

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

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

**al.add(60);**

**al.add(30);**

**al.add(50);**

**al.add(20);**

**al.add(70);**

**al.add(40);**

**al.add(80);**

**System.*out*.println("al : "+al);**

**//without stream**

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

**System.*out*.println(Arrays.*toString*(i));**

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

**}**

**}**

**/\* Stream object on array**

**\*/**

**import java.util.stream.Stream;**

**public class Test{**

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

**Stream s = Stream.*of*(11,22,333,4444,55555);**

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

**Integer[] i = {11,22,333,4444,55555};**

**Stream s1 = Stream.*of*(i);**

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

**}**

**}**

**/\* program on**

**\* iterate()**

**\* reduce()**

**\* summingDouble()**

**\***

**\*/**

**import java.util.ArrayList;**

**import java.util.List;**

**import java.util.stream.Collectors;**

**public class Test{**

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

**/\* Stream.iterate(1, (i->i+1)).filter(i->i%5==0).limit(5).**

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

**\*/**

**List<Integer> li = new ArrayList<Integer>();**

**li.add(10);**

**li.add(20);**

**li.add(30);**

**li.add(40);**

**/\*int total = li.stream().map(i->i).reduce(0,(sum,i)->sum+i);**

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

**int total1 = li.stream().map(i->i).reduce(0,Integer::sum);**

**System.out.println(total1);\*/**

**double total2 = li.stream().collect(Collectors.*summingDouble*(i->i));**

**System.*out*.println(total2);**

**}**

**}**

**Java Functional Composition:**

**import java.util.function.Predicate;**

**public class Test{**

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

**Predicate<String> p1 = (b) -> {**

**if(b.startsWith("R")){**

**return true;**

**}**

**else**

**return false;**

**};**

**Predicate<String> p2 = (b) -> {**

**if(b.endsWith("m")){**

**return true;**

**}**

**else**

**return false;**

**};**

**Predicate<String> p3 = (b) -> {**

**return p1.test(b) && p2.test(b);**

**};**

**System.*out*.println(p3.test("Ram"));**

**System.*out*.println(p3.test("RaM"));**

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

**Predicate<String> p4 = p1.and(p2);**

**System.*out*.println(p4.test("Ram"));**

**System.*out*.println(p4.test("RaM"));**

**System.*out*.println("==============\*\*\*==========");**

**Predicate<String> p5 = p1.or(p2);**

**System.*out*.println(p5.test("Ram"));**

**System.*out*.println(p5.test("RaM"));**

**}**

**}**

**import java.util.function.Function;**

**public class Test{**

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

**Function<Integer,Integer> f1 = (v)-> {return v+2;};**

**Function<Integer,Integer> f2 = (v)-> {return v\*3;};**

**//Function<Integer,Integer> f3 = f1.compose(f2);**

**Function<Integer,Integer> f3 = f1.andThen(f2);**

**System.*out*.println(f3.apply(10));**

**}**

**}**

**STREAM API:**

**--> It is the collection of classes and interface. It will provide some rules and implementation for processing, filtering, mapping, selecting objects from collection object....**

**--> It was launched in the version of JDK 1.8**

**--> Streams internally take support of functional interfaces, lambdas, method references....**

**--> Stream API is not related to IOSTREAM concept in java....**

**Stream Processing:**

**To perform some operations on elements or objects like filtering, mapping Stream API internally uses some objects those are called listeners. For processing each and every element Streams will uses one listener. Once one element processing by one listener, then control automatically handover to another listener for processing another element. That means internally streams maintaining some chain of listeners….**

**Basic ways of gets Stream Object:**

**Collection.parallelStream()**

**Collection.stream()**

**Arrays.stream(-)**

**Arrays.stream(-,-,-)**

**Stream.of(-)**

**Operations on Stream objects:**

**We have two types of operations belongs to Stream api.**

1. **Non-terminal operations**
2. **Terminal operations**

**Non-terminal Operations:**

**These are not disturbing the count of elements in a stream and finally these are return a stream object based on given condition.**

**Terminal operations:**

**These are iterating the each and every element by applying listeners and returns as result.**

**Program on map() method:[Non-Terminal operation] and count() method [Terminal operation]**

**import** java.util.ArrayList;

**import** java.util.stream.Stream;

**public** **class** Check{

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

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

al.add("ram");

al.add("sam");

al.add("ajay");

al.add("ranveer");

al.add("kaif");

System.***out***.println("al: "+al);

Stream<String> s = al.stream();

System.***out***.println("s: "+s);

//s.forEach((x)->{System.out.println(x);});

Stream<String> s1 = s.map((x)->x.toUpperCase());

//s1.forEach((x)->{System.out.println(x);});

**long** noOfElements = s1.count();

System.***out***.println("number of elements of a Stream: "+noOfElements);

}

}

In the above program if we uncomment any one of the comment we will get java.lang.IllegalStateExcepiton.

The reason is already Stream object is operated, that means one can apply one operation on Stream object, if want to apply multiple operations then we should maintain chain of operations.

**import** java.util.ArrayList;

**import** java.util.stream.Stream;

**public** **class** Check{

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

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

al.add("ramchandra");

al.add("samsundar");

al.add("ajaykumar");

al.add("ranveerkapoor");

al.add("yuvrajsingh");

System.***out***.println("al: "+al);

Stream<String> s = al.stream();

System.***out***.println("s: "+s);

Stream<String> s1 = s.map((x)->x.toUpperCase());

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

}

}

In the above program we count() on newly generated Stream object. If we want use both map() and count() at a time on Stream object then we should use bellow code.

**import** java.util.ArrayList;

**import** java.util.stream.Stream;

**public** **class** Check{

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

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

al.add("ramchandra");

al.add("samsundar");

al.add("ajaykumar");

al.add("ranveerkapoor");

al.add("yuvrajsingh");

System.***out***.println("al: "+al);

Stream<String> s = al.stream();

System.***out***.println("s: "+s);

System.***out***.println(s.map((x)->x.toUpperCase()).count());

}

}

Last statement is the combination of both terminal and non-terminal operations on Stream object

In the same manner we can apply multiple or chain of non-terminal of operations at a time like bellow.

**import** java.util.ArrayList;

**import** java.util.stream.Stream;

**public** **class** Check{

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

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

al.add("ramchandra");

al.add("samsundar");

al.add("ajaykumar");

al.add("ranveerkapoor");

al.add("yuvrajsingh");

System.***out***.println("al: "+al);

Stream<String> s = al.stream();

System.***out***.println("s: "+s);

Stream<String> s1 = s.map((x)->x.toUpperCase()).map((x)->x.substring(5));

s1.forEach((x) -> {System.***out***.println(x);});

}

}

Last but one statement will show chain of non-terminal operations on Stream object

**Program on filter() of Stream interface:**

**import** java.util.ArrayList;

**import** java.util.stream.Stream;

**public** **class** Check{

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

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

al.add(9);

al.add(20);

al.add(33);

al.add(44);

al.add(56);

al.add(60);

System.***out***.println("al: "+al);

Stream<Integer> s = al.stream();

System.***out***.println("s: "+s);

Stream<Integer> s1 = s.filter((x)-> {

**if**(x%2==0){

**return** **true**;

}

**else**

**return** **false**;

});

//System.out.println(s1.count());

s1.forEach((x) -> {System.***out***.println(x);});

}

}

**Program on floatMap():**

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.stream.Stream;

**public** **class** Check{

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

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

al.add("AA BB CC DD EE FF GG HH");

al.add("a b c d e f g h");

al.add("A B C D E F G H");

al.add("ram sam raj vas");

System.***out***.println("al: "+al);

Stream<String> s = al.stream();

System.***out***.println("s: "+s);

Stream<String> s1 = s.flatMap((x)->{

String ss = x;

String[] ss1 = ss.split(" ");

**return** Arrays.*asList*(ss1).stream();

});

//System.out.println(s1.count());

s1.forEach((x)-> {System.***out***.println(x);});

}

}

**Program on distinct():**

**import** java.util.ArrayList;

**import** java.util.stream.Stream;

**public** **class** Check{

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

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

al.add("one");

al.add("two");

al.add("three");

al.add("two");

al.add("one");

System.***out***.println("al: "+al);

Stream<String> s= al.stream();

Stream<String> s1 = s.distinct();

s1.forEach((x)->{System.***out***.println(x);});

}

}

**Program on limit():**

**import** java.util.ArrayList;

**import** java.util.stream.Stream;

**public** **class** Check{

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

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

al.add("one");

al.add("two");

al.add("three");

al.add("two");

al.add("one");

System.***out***.println("al: "+al);

Stream<String> s= al.stream();

//Stream<String> s1 = s.limit(4).distinct();

Stream<String> s1 = s.limit(4);

s1.forEach((x)->{System.***out***.println(x);});

}

}

**Program on peek():**

**import** java.util.ArrayList;

**import** java.util.stream.Stream;

**public** **class** Check{

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

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

al.add("one");

al.add("two");

al.add("three");

System.***out***.println("al: "+al);

Stream<String> s= al.stream();

Stream<String> s1 = s.peek((x)-> {System.***out***.println(x);});

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

}

}

**Program on sorted() of Stream Object:**

**package java8newfeatures;**

**import java.util.ArrayList;**

**import java.util.stream.Stream;**

**public class Test9{**

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

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

**al.add(30);**

**al.add(60);**

**al.add(20);**

**al.add(50);**

**al.add(10);**

**al.add(40);**

**System.out.println("al: "+al.size());**

**System.out.println("al: "+al);**

**Stream<Integer> s1 = al.stream();**

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

**//s1.forEach((x)->System.out.print(x+" "));**

**Stream<Integer> s2 = s1.sorted();**

**//System.out.println(s2.count());**

**s2.forEach((x)->System.out.print(x+" "));**

**}**

**}**

**Terminal Operations:**

**Program anyMatch(), allMatch(), noneMatch() method:**

**import** java.util.ArrayList;

**import** java.util.stream.Stream;

**public** **class** Check{

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

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

al.add("one");

al.add("two");

al.add("three");

System.***out***.println("al: "+al);

Stream<String> s= al.stream();

**boolean** s1 = s.anyMatch((x)->{

**return** x.startsWith("t");

});

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

Stream<String> s2= al.stream();

**boolean** s3 = s2.allMatch((x)->{

**return** x.startsWith("t");

});

System.***out***.println(s3);

Stream<String> s4= al.stream();

**boolean** s5 = s4.noneMatch((x)->{

**return** x.startsWith("o");

});

System.***out***.println(s5);

}

}

**Program on collect():**

**import** java.util.ArrayList;

**import** java.util.stream.Collectors;

**import** java.util.stream.Stream;

**public** **class** Check{

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

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

al.add("one");

al.add("two");

al.add("three");

System.***out***.println("al: "+al);

Stream<String> s= al.stream();

Stream<String> s1 = s.map((x) -> x.toUpperCase());

System.***out***.println(s1.collect(Collectors.*toList*()));

}

}

**Program on findAny() and findFirst() and forEach(-):**

**import** java.util.ArrayList;

**import** java.util.Optional;

**import** java.util.stream.Stream;

**public** **class** Check{

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

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

al.add("two");

al.add("one");

al.add("three");

System.***out***.println("al: "+al);

Stream<String> ss1= al.stream();

ss1.forEach((x)->{System.***out***.println(x);});

Stream<String> s= al.stream();

Optional<String> s1 = s.findAny();

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

Stream<String> ss= al.stream();

Optional<String> s2 = ss.findFirst();

System.***out***.println(s2.get());

}

}

**Program on max(), min(), reduce(), toArray():**

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.Optional;

**import** java.util.stream.Stream;

**public** **class** Check{

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

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

al.add("two");

al.add("one");

al.add("ome");

System.***out***.println("al: "+al);

Stream<String> ss1= al.stream();

Optional<String> o1 = ss1.min((x,y)->{

**return** x.compareTo(y);

});

System.***out***.println(o1.get());

Stream<String> ss2= al.stream();

Optional<String> o2 = ss2.max((x,y)->{

**return** x.compareTo(y);

});

System.***out***.println(o2.get());

Stream<String> ss3= al.stream();

Optional<String> o3 = ss3.reduce((x,y)->{

**return** y+x;

});

System.***out***.println(o3.get());

Stream<String> ss4= al.stream();

Object[] o = ss4.toArray();

System.***out***.println(Arrays.*toString*(o));

}

}

**Concatanation of Streams:**

**import** java.util.ArrayList;

**import** java.util.stream.Stream;

**public** **class** Check{

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

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

al.add("two");

al.add("one");

al.add("ome");

System.***out***.println("al: "+al);

Stream<String> ss= al.stream();

ArrayList<String> al1 = **new** ArrayList<String>();

al1.add("2");

al1.add("1");

al1.add("3");

System.***out***.println("al: "+al1);

Stream<String> ss1= al1.stream();

Stream<String> s = Stream.*concat*(ss, ss1);

s.forEach((x)->{System.***out***.println(x);});

}

}

**Program on Stream.of(-):**

**import** java.util.stream.Stream;

**public** **class** Check{

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

Stream<String> s =Stream.*of*("1","2","3");

s.forEach((x)->{System.***out***.println(x);});

}

}

**Program on difference between sream() and parallelStream()**

**import** java.time.LocalTime;

**import** java.util.Arrays;

**import** java.util.stream.Stream;

**public** **class** Check{

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

String[] strings = {"1", "2", "3", "4", "5", "6", "7", "8", "9", "10"};

System.***out***.println("-------\nRunning sequential\n-------");

*run*(Arrays.*stream*(strings).sequential());

System.***out***.println("-------\nRunning parallel\n-------");

*run*(Arrays.*stream*(strings).parallel());

}

**public** **static** **void** run (Stream<String> stream) {

stream.forEach(s -> {

System.***out***.println(LocalTime.*now*() + " - value: " + s +

" - thread: " + Thread.*currentThread*().getName());

**try** {

Thread.*sleep*(200);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

});

}

}

**Program on Arrays.stream(-):**

**import** java.util.Arrays;

**import** java.util.stream.Stream;

**public** **class** Check{

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

String[] strings = {"1", "2", "3", "4", "5", "6", "7", "8", "9", "10"};

Stream<String> s = Arrays.*stream*(strings);

Object[] o = s.toArray();

System.***out***.println(Arrays.*toString*(o));

}

}

**Program on Arrays.stream(-,-,-):**

**import** java.util.Arrays;

**import** java.util.stream.Stream;

**public** **class** Check{

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

String[] strings = {"1", "2", "3", "4", "5", "6", "7", "8", "9", "10"};

Stream<String> s = Arrays.*stream*(strings, 2, 5);

Object[] o = s.toArray();

System.***out***.println(Arrays.*toString*(o));

}

}

**import java.util.ArrayList;**

**import java.util.List;**

**import java.util.Optional;**

**import java.util.stream.Stream;**

**public class Test{**

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

**List<String> stringList = new ArrayList<String>();**

**/\*stringList.add("4 2 3 4 5 6");**

**stringList.add("2 8 9 0 1 2");**

**stringList.add("3 2 3 4 5 6");\*/**

**/\*Stream<String> stream = stringList.stream();**

**System.out.println(stream.count());\*/**

**/\*Stream<String> stringStream =**

**stream.map((value) -> { return value.toLowerCase(); });**

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

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

**\*/**

**//Stream<String> stream = stringList.stream();**

**/\*Stream<String> stream1 = stream.map((value)-> {return value.toLowerCase();});**

**stream1.forEach(System.out::println);\*/**

**/\*Stream<String> stream1 =**

**stream.map((value)-> {return value.toLowerCase();})**

**.map((value)-> {return value.toUpperCase();});**

**stream1.forEach(System.out::println);**

**\*/**

**/\*Stream<String> longStringsStream = stream.filter((value) -> {**

**return value.length() >= 3;**

**});**

**longStringsStream.forEach(System.out::println);\*/**

**/\*Stream<String> stream = stringList.stream();**

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

**Stream<Object> s2 = stream.flatMap((value)->{**

**String[] s1 = value.split(" ");**

**return Arrays.asList(s1).stream();**

**});**

**//s2.forEachOrdered(System.out::println);**

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

**System.out.println(s2.count());\*/**

**//Stream<String> stream = stringList.stream().distinct();**

**//Stream<String> stream = stringList.stream().limit(2);**

**/\*Stream<String> stream = stringList.stream().peek((value)->{**

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

**});**

**stream.forEachOrdered(System.out::println);\*/**

**/\*int count = (int) stringList.stream().count();**

**System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* : "+count);**

**\*/**

**/\*Stream<String> stream = stringList.stream();**

**boolean b = stream.anyMatch((value)->{ return value.startsWith("1");**

**});**

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

**/\*Stream<String> stream = stringList.stream();**

**boolean b = stream.allMatch((value)->{ return value.startsWith("1");**

**});**

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

**/\*Stream<String> stream = stringList.stream();**

**boolean b = stream.noneMatch((value)->{ return value.startsWith("1");**

**});**

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

**stringList.add("suji");**

**stringList.add("ram");**

**stringList.add("sam");**

**stringList.add("annie");**

**stringList.add("xam");**

**Stream<String> stream = stringList.stream();**

**//Optional<String> os = stream.findAny();**

**//Optional<String> os = stream.findFirst();**

**/\*Optional<String> os = stream.max((v1,v2)->{**

**return v1.compareTo(v2);**

**});\*/**

**Optional<String> os = stream.min((v1,v2)->{**

**return v1.compareTo(v2);**

**});**

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

**/\***

**import java.util.ArrayList;**

**import java.util.Arrays;**

**import java.util.List;**

**import java.util.Optional;**

**import java.util.stream.Collectors;**

**import java.util.stream.Stream;**

**class Test{**

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

**/\*List<String> li = new ArrayList<String>();**

**li.add("varun kumar");**

**li.add("ramsa chndra");**

**li.add("mahi kumar");**

**li.add("suresh chandra");\*/**

**//System.out.println(s.count());**

**/\*System.out.println(s.anyMatch((x)->{**

**return x.startsWith("ram");**

**}));\*/**

**/\*System.out.println(s.allMatch((x)->{**

**return x.startsWith("ram");**

**}));\*/**

**/\*System.out.println(s.noneMatch((x)->{**

**return x.startsWith("ram");**

**}));\*/**

**/\*System.out.println(s.anyMatch((x)->{**

**return x.startsWith("ram");**

**}));\*/**

**/\*Optional<String> os =s.findAny();**

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

**\*/**

**/\*Optional<String> os =s.findFirst();**

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

**\*/**

**/\*s.forEach(System.out::println);**

**s.close();**

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

**Stream<String> s1 = li.stream();**

**s1.forEach((value)->{System.out.println(value);});**

**Stream<String> s2 = li.stream();**

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

**s2.forEachOrdered((value)->{System.out.println(value);});\*/**

**List<String> li = new ArrayList<String>();**

**li.add("A");**

**li.add("B");**

**li.add("C");**

**li.add("D");**

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

**Stream<String> s = li.stream();**

**/\*Optional<String> os = s.max((v1,v2)->{return v1.compareTo(v2);});**

**System.out.println(os.get());\*/**

**/\*Optional<String> os = s.min((v1,v2)->{return v1.compareTo(v2);});**

**System.out.println(os.get());\*/**

**/\*Optional<String> os = s.reduce((v1,v2)->{return v1+v2;});**

**System.out.println(os.get());\*/**

**/\*Object[] o = s.toArray();**

**//System.out.println(Arrays.toString(o));**

**for(Object o1: o){**

**System.out.println(o1);\*/**

**List<String> li1 = new ArrayList<String>();**

**li1.add("11");**

**li1.add("22");**

**li1.add("33");**

**li1.add("44");**

**System.*out*.println(li1);**

**Stream<String> s1 = li1.stream();**

**Stream<String> s2 = Stream.*concat*(s, s1);**

**List<String> s3 = s2.collect(Collectors.*toList*());**

**System.*out*.println(s3);**

**}**

**}**

**\*/**

**Stream.*of*("aaa","bbb","ccc").**

**parallel().forEach((ss)->{System.*out*.println(ss);});**

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

**Stream.*of*("aaa","bbb","ccc").**

**parallel().forEachOrdered((ss)->{System.*out*.println(ss);});**

**}**

**}**

Functional programming in java:

Key elements in Functional programming:

1. Functions as first class objects.

@FunctionalInterface

**interface** I{

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

}

**public** **class** Check {

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

I obj = ()->System.***out***.println("lambdas");

obj.m1();

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

}

}

1. Pure functions
2. No side efforts
3. Based on input only output will come

@FunctionalInterface

**interface** I{

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

}

**public** **class** Check {

**static** **int** *a* = 111;

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

I obj = (l,b)->{

**return** (l+b);

};

System.***out***.println(obj.m1(10,20));

}

}

Above program satisfies the rules 1 and 2.

But bellow program not satisfies the above rules.

**@FunctionalInterface**

**interface I{**

**public abstract String m1(int a, int b);**

**}**

**public class Check {**

**static int *a* = 111;**

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

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

**I obj = (l,b)->{**

***a*=*a*+l+b;**

**return "the result is: "+*a*;**

**};**

**System.*out*.println(obj.m1(10,20));**

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

**}**

**}**

**Higher order functions:**

**It takes one or more parameters as functions.**

**Return type of the function is also one function.**

**@FunctionalInterface**

**interface I{**

**public abstract int m1();**

**}**

**@FunctionalInterface**

**interface J{**

**public abstract int m2();**

**}**

**@FunctionalInterface**

**interface K{**

**public abstract int m3(I obj1, J obj2);**

**}**

**public class Check {**

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

**I obj1 = ()-> {return 111;};**

**J obj2 = ()-> {return 222;};**

**K obj = (o1,o2)->{**

**return o1.m1()+o2.m2();**

**};**

**System.*out*.println(obj.m3(obj1, obj2));**

**}**

**}**

**No State:**

**@FunctionalInterface**

**interface I{**

**public abstract void m1(int x);**

**}**

**public class Check {**

**int a = 111;**

**public void check(){**

**I obj = (y)->{**

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

**};**

**obj.m1(222);**

**}**

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

**Check obj = new Check();**

**obj.check();**

**}**

**}**

**Above violates no state rule.**

**No side effects.**

**Contains immutable variables.**

**Favor recursion over looping.**

**StringJoiner:**

**Collectors:**

**import java.util.ArrayList;**

**import java.util.List;**

**import java.util.Set;**

**import java.util.stream.Collectors;**

**class Product{**

**int id;**

**String name;**

**double price;**

**public Product(int id, String name, double price) {**

**super();**

**this.id = id;**

**this.name = name;**

**this.price = price;**

**}**

**@Override**

**public String toString(){**

**return id+".."+name+".."+price;**

**}**

**}**

**public class Check {**

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

**List<Product> l = new ArrayList<Product>();**

**Product p1 = new Product(1,"cj",500);**

**Product p2 = new Product(2,"aj",600);**

**Product p3 = new Product(3,"pj",400);**

**Product p4 = new Product(4,"dj",800);**

**Product p5 = new Product(5,"ej",200);**

**l.add(p1);**

**l.add(p2);**

**l.add(p3);**

**l.add(p4);**

**l.add(p5);**

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

**/\*List<Double> l1 = l.stream().map((p)->p.price).**

**collect(Collectors.toList());**

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

**/\*Set<Double> l1 = l.stream().map((p)->p.price).**

**collect(Collectors.toSet());**

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

**/\*double d = l.stream()**

**.collect(Collectors.averagingDouble((p)->p.price));**

**System.out.println(d);\*/**

**/\*Double i =l.stream().collect(Collectors.averagingInt((p)->p.id));**

**System.out.println(i);\*/**

**/\*int j =l.stream().collect(Collectors.summingInt((p)->p.id));**

**System.out.println(j);\*/**

**/\*Long k = l.stream().collect(Collectors.counting());**

**System.out.println(k);\*/**

**double d1 = l.stream().**

**collect(Collectors.*summingDouble*((p)->p.price));**

**System.*out*.println(d1);**

**DoubleSummaryStatistics is= l.stream().**

**collect(Collectors.*summarizingDouble*((p)->p.price));**

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

**IntSummaryStatistics is1= l.stream().**

**collect(Collectors.*summarizingInt*((p)->p.id));**

**System.*out*.println(is1);**

**}**

**}**

String[][] ss = {{"cj","ram"},{"aj","sam"},{"oracle","vijay"}};

Stream<String[]> ss1 = Stream.*of*(ss);

Map<String,String> m = ss1.collect(Collectors.*toMap*(x->x[0], x->x[1]));

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

**Type Inference:**

**package** java8newfeatures;

**import** java.util.ArrayList;

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

**static** **public** **void** printDetails(ArrayList<Integer> al){

**boolean** b = al.isEmpty();

**if**(!b){

al.forEach((x)->System.***out***.println(x));

}

**else**{

System.***out***.println("arraylist is empty");

}

}

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

//before jdk 7.0

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

al.add(10);

al.add(20);

al.add(30);

al.add(40);

Test.*printDetails*(al);

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

//in jdk 7.0

ArrayList<Integer> al1 = **new** ArrayList<>();

al1.add(10);

al1.add(20);

al1.add(30);

al1.add(40);

Test.*printDetails*(al1);

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

//from jdk 8.0

Test.*printDetails*(**new** ArrayList<>());

}

}

Method Parameter Reflection:

Calculation.java:

**package** java8newfeatures;

**public** **class** Calculation {

**int** add(**int** a, **int** b){

**return** a+b;

}

**int** mul(**int** a, **int** b){

**return** a\*b;

}

}

**Compile above program like bellow:**

**Javac –parameters Calculation.java**

**::Test.java::**

**package** java8newfeatures;

**import** java.lang.reflect.Method;

**import** java.lang.reflect.Parameter;

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

**public** **static** **void** main(String []x)**throws** ClassNotFoundException,

InstantiationException, IllegalAccessException{

Class cls = Class.*forName*("java8newfeatures.Calculation");

Object obj = cls.newInstance();

Method[] m = cls.getDeclaredMethods();

**for**(Method m1: m){

System.***out***.println(m1);

System.***out***.println(m1.getName());

System.***out***.println(m1.getParameterCount());

Parameter[] p = m1.getParameters();

**for**(Parameter p1: p){

System.***out***.println("\t"+p1);

System.***out***.println("\t"+p1.getName());

System.***out***.println("\t"+p1.getModifiers());

//System.out.println("\t"+p1.hashCode());

System.***out***.println("\t"+p1.getParameterizedType());

System.***out***.println("\t"+p1.getName());

}

}

}

}

Compilation: javac Test.java

Execution: java java8newfeatures.Test