**Lambda Expression**

Lambda Expression is just an anonymous (name-less) function.

That means the function which doesn't have the name, return type and access modifiers.

Lambda expression is also known an anonymous functions or clousers.

Example1:

*public void printHello(){*

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

*}*

convert the above normal method into lambda expression

*() -> {*

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

*}*

if only one statement is there then no need to specify { }

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

Example2:

purblic void printMessage(String msg){

System.out.println(msg);

}

(String msg) -> { System.out.println(msg); }

no need specifies the perameter type.

(msg) -> { System.out.println(msg); }

(msg) -> System.out.println(msg);

if only one parameter is there then no need to write parenthesis.

msg -> System.out.println(msg);

Example3:

public int sum(int a, int b){

return a+b;

}

convert the above normal method into lambda expression

(int a, int b) -> {

return a+b;

};

if the type of the parameter can be decided by the compiler automatically based on the context then we can remove parameter types.

(a, b) -> { return a+b; }

if the compiler can decide that as there is only one line and that is return statement then we can remove return key word.

(a,b) -> { a+b };

(a,b) -> a+b;

a -> a\*a;

**We understood the below from above examples:**

1. a lambda expression can have zero or more number of parameters (or arguments), parameters are comma separated

2. no need to specify parameter types

3. if only one line and that is return statement then no use return keyword

4. if zero number of parameters are there then we can have to use only empty parenthesis ()

5) if only parameter is there then we can remove parenthesis

6) like a method lambda expression can also have multiple statement if multiple statements are there then { and } is compulsory.

7) Once we write lambda expression, we can call the expression just like a method, for this functional interfaces are required.

**Functional Interface**

An interface contains only one abstract method such type of interface is called functional interface and the method is called functional method or single abstract method.

for functional interface method we can assign a lambda expression.

Examples of predefined functional interfaces are:

1) Runnable interface has only one abstract method public void run()

2) Callable interface has only one abstract method public V call();

3) Comparator interface has only one abstract method public int compareTo(T t)

To provide definitions for Runnable interface we can define class implements runnable interface

*public class MyRunner implements Runnable{*

*public void run(){*

*System.out.println("run method definition");*

*}*

*}*

…….

*Runnable r1 = new MyRunner();*

*Thread t1 = new Thread(r1);*

*t1.start(); // it will call run method*

To provide definitions for Runnable interface we can define anonymous classes

*Runnable r2 = new Runnable(){*

*public void run() {*

*System.out.println("run method definition");*

*}*

*};*

*Thread t2 = new Thread(r2);*

*t2.start(); // it will call run method*

To provide definitions for Runnable interface run method we can define lambda expression as well

*Runnable r3 = () -> System.out.println("run method definition");*

*Runnable r4 = () -> {*

*System.out.println("run method definition statement1");*

*System.out.println("run method definition statement2");*

*};*

*Thread t3 = new Thread(r3);*

*t3.start(); // it will call run method*

User Defined Example of functional interface:

*interface ArithmeticOperator{*

*int perform(int a, int b);*

*}*

implementation by classes

*class AdditionOp implements ArithmeticOperator{*

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

*return a+b;*

*}*

*}*

*ArithmeticOperator op1 = new AdditionOp();*

*int c = op1.perform(2,3);*

*class SubtractionOp implements ArithmeticOperator{*

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

*return a-b;*

*}*

}

*ArithmeticOperator op2 = new SubtractionOp();*

*int c = op2.perform(2,3);*

*ArithmeticOperator add = new ArithmeticOperator(){*

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

*return a+b;*

*}*

*};*

*add.peform(2,2);*

here in the above example classes are defined and methods are defined.

implementation by lambda expression:

ArithmeticOperator addition = (p,q) -> p+q;

ArithmeticOperator subtraction = (m,n) -> m-n;

int result1 = addition.perform(2,3); for easy understanding we can assume java compiler will convert as int result1 = 2+3;

int result2 = subtraction.perform(5,3); java compiler will convert as int result2 = 5-3;

but in this example no need to define a separate for simple and single expression instead we can use lambda expression.

Note:

Inside a functional interface along with abstract method declaration we can also define any number default methods and static methods.

**What is the use @FunctionalInterface?**

In java 8, we have to use @FunctionalInterface annotations to specify that our interface is a Funcational inteface and this annotation avoids accidental addition of new methods to our interface in future releases.

*package com.satya.newutils;*

*public interface ArithmeticOperator{*

*int perform(int a, int b);*

*}*

rest of the application is using newutils and its interfaces.

after couple of months new developer joined in the team he want to update ArithmeticOperator by adding one method then it is possible.

once it is updated with one more abstract method then it will not be a function interface.

Because of this our application code which is having lambda expressions will be disturbed.

So to avoid that it is recommended to use @FunctionalInterface annotation:

*package com.satya.newutils;*

*@FunctionalInterface*

*public interface ArithmeticOperator{*

*int perform(int a, int b);*

*}*

Note: @FunctionalInterface annotation lets us to add only one abstract method in the interface.

if we accidentally add one more method, we will get compilation error.

**Functional interface with respect to inheritance**

Point1: If a child interface extends functional interface and if that child interface does not contain any abstract methods then the child interface is also FunctionalInterface by default.

*@FunctionalInterface*

*interface A{*

*public void methodOne();*

*}*

*@FuncationalInterface*

*interface B extends A{*

*}*

The above code is allowed no compilation errors

Point2: In the child interface we can write exactly same parent interface abstract method but not new abstract methods.

*@FunctionalInterface*

*interface A{*

*public void methodOne();*

*}*

*@FuncationalInterface*

*interface B extends A{*

*public void methodOne();*

*}*

The above code is allowed no compilation errors

*@FunctionalInterface*

*interface A{*

*public void methodOne();*

*}*

*@FuncationalInterface*

*interface B extends A{*

*public void methodOne(int a);*

*}*

The above code is not allowed we will get compilation errors

error: Unexpected @FunctionalInterface annotation

@FunctionalInterface

^

B is not a functional interface

multiple non-overriding abstract methods found in interface B

1 error

as per this error we cannot make B as functional interface.

*@FunctionalInterface*

*interface A{*

*public void methodOne();*

*}*

*@FuncationalInterface*

*interface B extends A{*

*public void methodTwo();*

*}*

This also leads error

**Functional interface VS Lambda Expressions**

To invoke a lambda expression functional interface is required.

Functional interface references are used to refer Lambda expressions.

Whenever we are assigning a lambda expression to a functional interface reference it means we are pointing methods by using references.

*interface Pinter{*

*public abstract void printMessage(String msg);*

*}*

.....

*Printer p = (m) -> System.out.println("message");*

*p.printMessage("satya");*

*p.printMessage("ravi");*

*p.printMessage("surya");*

here Pinter is functional interface by default as it is having only one abstract method.

we can annotate Printer with @FunctionalInterface but it is not mandatory.

here we are executing the lambda expression three times.

here p is used to refer lambda expression.

p is pointing method(p is reference to method)

we can execute that method by using p.printMessage("....");

**Comparisons between Anonymous inner class and Lambda expressions**

When ever we are using anonymous inner classes there may be a chance of using lambda expression to reduce length of the code and to resolve complexity.

Summarize the advantages of lambda expressions:

* we can reduce length of the code so that readability of the code will be improved
* we can resolve complexity of anonymous inner class.
* we can provide lambda expression in the place of object.
* as we can pass object references as argument so we can also pass lambda expressions as arguments to methods.
* as we can return object references so we can also return lambda expression as return values from methods.

**Practice Exercise**

**interface** CalculatorOperation {

**int** perform(**int** a, **int** b);

}

**class** TwoOperands {

**int** x;

**int** y;

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

x = a;

y = b;

}

// we can pass lambda expression as method parameters

**public** **int** compute(CalculatorOperation operation) {

**return** operation.perform(x, y);

}

}

**public** **class** LEAsMethodParams{

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

TwoOperands twoOperands = **new** TwoOperands(6, 4);

CalculatorOperation additionOperation = **new** CalculatorOperation() {

@Override

**public** **int** perform(**int** x, **int** y) {

**return** x + y;

}

};

CalculatorOperation subtractionOperation = **new** CalculatorOperation() {

@Override

**public** **int** perform(**int** x, **int** y) {

**return** x - y;

}

};

CalculatorOperation multiplicationOperation = **new** CalculatorOperation() {

@Override

**public** **int** perform(**int** x, **int** y) {

**return** x \* y;

}

};

System.***out***.println(twoOperands.compute(additionOperation));

System.***out***.println(twoOperands.compute(subtractionOperation));

System.***out***.println(twoOperands.compute(multiplicationOperation));

}

}

/\*

public class LEAsMethodParams{

public static void main(String[] args){

TwoOperands twoOperands = new TwoOperands(6, 4);

System.out.println(twoOperands.compute((x,y)->x+y));

System.out.println(twoOperands.compute((x,y)->x-y));

System.out.println(twoOperands.compute((x,y)->x\*y));

}

}

\*/

LambdaAsReturnValueDemo.java

**package** com.satya1;

**interface** ArithMeticOperation {

**int** compute();

}

//this class is having methods returning lambda expression

**class** MathCalulator {

**public** ArithMeticOperation multiplicationOperation(**int** a, **int** b) {

ArithMeticOperation m = **new** ArithMeticOperation() {

@Override

**public** **int** compute() {

**return** a \* b;

}

};

**return** m;

// return () -> a\*b;

}

**public** ArithMeticOperation additionOperation(**int** a, **int** b) {

**return** () -> a + b;

}

**public** ArithMeticOperation subtractionOperation(**int** a, **int** b) {

**return** () -> a - b;

}

**public** ArithMeticOperation divisionOperation(**int** a, **int** b) {

**return** () -> a / b;

}

**public** ArithMeticOperation negation(**int** a) {

// outer method variables are final to lambda expression we can use but we can

// not modify them

**return** () -> {

// a=5;//error

System.***out***.println("do nothing negating a:" + a);

**return** -a;

};

}

}

**public** **class** LambdaAsReturnValue {

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

MathCalulator calc = **new** MathCalulator();

ArithMeticOperation negation1 = calc.negation(23);

ArithMeticOperation negation2 = calc.negation(-23);

System.***out***.println(negation1.compute());// -23

System.***out***.println(negation2.compute());// 23

System.***out***.println(calc.negation(55).compute());// -55

System.***out***.println(calc.additionOperation(2, 3).compute());

System.***out***.println(calc.multiplicationOperation(2, 3).compute());

}

}

**package** com.satya.java8ex.lambda;

//Lambdas vs Interface implementations

**interface** Printer {

**public** **abstract** **void** print(String msg);

}

**interface** Tester {

**public** **abstract** **void** test();

}

**public** **class** LambdaTest1 {

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

Tester tester1 = () -> System.***out***.println("Testing done by tester1");

Tester tester2 = () -> {

System.***out***.println("Testing done by tester2 in line1");

System.***out***.println("Testing done by tester2 in line2");

};

tester1.test();

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

tester1.test();

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

tester2.test();

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

Printer printer1, printer2;

printer1 = **new** Printer() {

**public** **void** print(String msg) {

System.***out***.println("printing " + msg + "message done! by Anonymous innter class");

}

};

printer2 = (String msg) -> System.***out***.println("printing " + msg + "message done! by Lambda expression");

printer1.print("1st message");

printer1.print("2nd message");

printer2.print("3rd message");

printer2.print("4th message");

}

}

**package** com.satya.java8ex.lambda;

**public** **class** LambdaTest2 {

**static** Printer *epsonPrinter* = (m) -> System.***out***.println("Starting printring the message:"+m+" on epson printer");

**static** Printer *canonPrinter* = (String m) -> System.***out***.println("started printing...."+m+" on canon printer ");

**static** Printer *greetingPrinter* = m -> System.***out***.println("Hello!:"+m);

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

*doPrinting*((m)->System.***out***.println("doing printing the message:"+m),"Hellow World!");

*doPrinting*(*epsonPrinter*,"Hello! How are you?");

*doPrinting*(*epsonPrinter*,"Hello! How do you do?");

*doPrinting*(*canonPrinter*, "This bond james bond");

*doPrinting*(*greetingPrinter*, "satya");

*doPrinting*(m -> System.***out***.println("Hello!:"+m), "deepak");

}

**public** **static** **void** doPrinting(Printer p,String msg) {

p.print(msg);

}

}

**package** com.satya.java8ex.lambda;

//Type Inference

@FunctionalInterface

**interface** Calculator {

**int** compute(**int** a, **int** b);

}

**public** **class** LambdaTest3 {

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

Calculator add1 = (**int** a, **int** b) -> {

**return** a + b;

};

Calculator add2 = (**int** a, **int** b) -> a + b; // type inference

Calculator add3 = (a, b) -> a + b; // type inference

Calculator subtraction = (a, b) -> {

**return** a - b;

};

Calculator multiPlication = (a, b) -> a \* b; // type inference

**int** x = add1.compute(2, 3);

**int** y = add2.compute(7, 3);

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

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

*executeOperation*(add3, 1, 3);

*executeOperation*(subtraction, 6, 3);

*executeOperation*(multiPlication, 7, 8);

*executeOperation*(multiPlication, 9, 2);

*executeOperation*((a, b) -> a / b, 9, 3);// type inference

}

**public** **static** **void** executeOperation(Calculator operation, **int** operand1, **int** operand2) {

**int** result = operation.compute(operand1, operand2);

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

}

}

**package** com.satya.java8ex.lambda;

/\*

\* (in java api all existing interfaces with only one method supports lambda)

\* example: Runnable Using Lambdas

\*/

**public** **class** LambdaTest4 {

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

Runnable r1 = **new** Runnable() {

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

System.***out***.println("Runnable1 run");

}

};

Runnable r2 = () -> System.***out***.println("Runnbale2 run");

Thread t1 = **new** Thread(r1);// passing impl ref

Thread t2 = **new** Thread(r2); // passing fun ref

Thread t3 = **new** Thread(() -> System.***out***.println("Runnbale3 run")); // same as above

t1.start();

t2.start();

t3.start();

}

}

/\*\*

\*

\* **@author** SATYA Functional interface. It is an interface with only one abstract

\* method but any number of default methods and static methods. It is

\* good practice to add **@FunctionalInterface** to such interface to avoid

\* accidentally adding new abstract methods(in the future).

\*

\*/

**package** com.satya.java8ex.lambda;

@FunctionalInterface

**interface** DocWriter {

**public** **abstract** **void** writeDoc(String msg);

**public** **default** **void** instanceMethod() {

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

}

**public** **static** **void** staticMethod() {

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

}

}

**public** **class** LambdaTest5 {

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

String message = "New Feature";

DocWriter docWriter = m -> System.***out***.println("writing:" + m + " to document done.");

docWriter.writeDoc("Update the feature");

*writeToDocument*(docWriter, message);

docWriter.instanceMethod();

DocWriter.*staticMethod*();

}

**public** **static** **void** writeToDocument(DocWriter docWriter, String msg) {

docWriter.writeDoc(msg);

}

}

**package** com.satya.java8ex.lambda;

**import** java.io.IOException;

**import** java.io.PrintStream;

**import** java.util.Arrays;

**import** java.util.Comparator;

**import** java.util.List;

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

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

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

**class** Resource {

**private** **int** id;

**private** String name;

**private** String role;

**public** Resource() {

}

**public** Resource(**int** id, String name, String department) {

**this**.id = id;

**this**.name = name;

**this**.role = department;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

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

**this**.name = name;

}

**public** String getRole() {

**return** role;

}

**public** **void** setRole(String role) {

**this**.role = role;

}

**public** String toString() {

**return** id + "," + name + "," + role;

}

}

@FunctionalInterface

**interface** Condition {

**public** **abstract** **boolean** isOk(Resource r);

}

**public** **class** LambdaTest6 {

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

// testCase1();

*testCase2*();

// testCase3();

// testCase4();

// testCase5();

// testCase6();

}

**public** **static** **void** testCase1() {

List<Resource> resourcesList = Arrays.*asList*(**new** Resource(23, "satya", "admin"),

**new** Resource(13, "rahul", "systemadmin"), **new** Resource(53, "ashok", "engineer"),

**new** Resource(93, "deepak", "sr-manager"), **new** Resource(33, "durga", "manager"));

*display*(resourcesList);

/\*\*\*\*\*\*\*\*\* with out lambdas \*\*\*\*\*\*\*\*\*\*\*\*\*/

// step1: sort list by name

/\*

\* Comparator nameComparator = new Comparator<Resource>() {

\*

\* @Override public int compare(Resource e1, Resource e2) { return

\* e1.getName().compareTo(e2.getName()); } };

\*

\* resourcesList.sort(nameComparator);

\*/

resourcesList.sort(**new** Comparator<Resource>() {

@Override

**public** **int** compare(Resource e1, Resource e2) {

**return** e1.getName().compareTo(e2.getName());

}

});

// step2: create a method that prints all elements in the list

*display*(resourcesList);

// step3: create a method that prints all people that have name begining with d

*display*(resourcesList, **new** Condition() {

@Override

**public** **boolean** isOk(Resource r) {

**return** r.getName().startsWith("d");

}

});

}

**public** **static** **void** testCase2() {

List<Resource> resources = Arrays.*asList*(**new** Resource(23, "satya", "admin"),

**new** Resource(13, "arahul", "systemadmin"), **new** Resource(53, "ashok", "engineer"),

**new** Resource(93, "deepak", "sr-manager"), **new** Resource(33, "durga", "manager"));

*display*(resources);

/\*\*\*\*\*\*\*\*\* with lambdas \*\*\*\*\*\*\*\*\*\*\*\*\*/

// step1: sort list

// by name

resources.sort((e1, e2) -> e1.getName().compareTo(e2.getName()));

// step2: create a method that prints all elements in the list

*display*(resources);

// sort list by role

resources.sort((e1, e2) -> e1.getRole().compareTo(e2.getRole()));

*display*(resources);

// sort list by id

resources.sort((e1, e2) -> e1.getId() - e2.getId());

*display*(resources);

// step3: create a method that prints all people that have name begining with C

*display*(resources, r -> r.getName().startsWith("d"));

*display*(resources, r -> r.getName().startsWith("a"));

}

**public** **static** **void** testCase3() {

List<Resource> resources = Arrays.*asList*(**new** Resource(23, "satya", "admin"),

**new** Resource(13, "rahul", "systemadmin"), **new** Resource(53, "ashok", "engineer"),

**new** Resource(93, "deepak", "sr-manager"), **new** Resource(33, "durga", "manager"));

/\*\*\*\*\*\*\*\*\* with lambdas \*\*\*\*\*\*\*\*\*\*\*\*\*/

// step1: sort list by name

resources.sort((e1, e2) -> e1.getName().compareTo(e2.getName()));

// step2: create a method that prints all elements in the list

*display*(resources, r -> **true**);

// step3: create a method that prints all people that have name begining with d

*display*(resources, r -> r.getName().startsWith("d"));

// step4: create a method that prints all people that have role name

// contians with admin

*display*(resources, r -> r.getRole().contains("admin"));

}

// working with java.util.function.Predicate

**public** **static** **void** testCase4() {

List<Resource> resources = Arrays.*asList*(**new** Resource(23, "satya", "admin"),

**new** Resource(13, "rahul", "systemadmin"), **new** Resource(53, "ashok", "engineer"),

**new** Resource(93, "deepak", "sr-manager"), **new** Resource(33, "durga", "manager"));

/\*\*\*\*\*\*\*\*\* with lambdas \*\*\*\*\*\*\*\*\*\*\*\*\*/

// step1: sort list by name

resources.sort((e1, e2) -> e1.getName().compareTo(e2.getName()));

// step2: create a method that prints all elements in the list

*displayBasedOnPredicate*(resources, r -> **true**);

// step3: create a method that prints all people that have name begining with d

*displayBasedOnPredicate*(resources, r -> r.getName().startsWith("d"));

// step4: create a method that prints all people that have department name

// contians with admin

*displayBasedOnPredicate*(resources, r -> r.getRole().contains("admin"));

}

// working with java.util.function

**public** **static** **void** testCase5() {

List<Resource> resources = Arrays.*asList*(**new** Resource(23, "satya", "admin"),

**new** Resource(13, "rahul", "systemadmin"), **new** Resource(53, "ashok", "engineer"),

**new** Resource(93, "deepak", "sr-manager"), **new** Resource(33, "durga", "manager"));

/\*\*\*\*\*\*\*\*\* with lambdas \*\*\*\*\*\*\*\*\*\*\*\*\*/

// step1: sort list by name

resources.sort((e1, e2) -> e1.getName().compareTo(e2.getName()));

// step2: create a method that prints all elements in the list

*displayBasedOnPredicate*(resources, r -> **true**, **new** Consumer<Resource>() {

**public** **void** accept(Resource r) {

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

}

});

*displayBasedOnPredicate*(resources, r -> **true**, r -> System.***out***.println(r));

*displayBasedOnPredicate*(resources, p -> **true**, System.***out***::println);

**try** {

PrintStream ps = **new** PrintStream("c:\\temp\\s.txt");

*displayBasedOnPredicate*(resources, r -> **true**, r -> ps.println(r));

*displayBasedOnPredicate*(resources, r -> **true**, ps::println);

ps.close();

} **catch** (IOException e) {

e.printStackTrace();

}

// step3: create a method that prints all people that have name begining with C

*displayBasedOnPredicate*(resources, r -> r.getName().startsWith("d"), r -> System.***out***.println(r));

// step4: create a method that prints all people that have department name

// contians with admin

*displayBasedOnPredicate*(resources, r -> r.getRole().contains("admin"), r -> System.***out***.println(r));

*displayBasedOnPredicate*(resources, r -> r.getRole().contains("admin"), r -> System.***out***.println(r.getName()));

*displayBasedOnPredicate*(resources, r -> r.getRole().contains("admin"), r -> System.***out***.println(r.getRole()));

*displayBasedOnPredicate*(resources, r -> r.getRole().contains("admin"), r -> System.***out***.println(r.getId()));

}

**public** **static** **void** testCase6() {

Function<String, Integer> lengthFun = s -> s.length();

System.***out***.println(lengthFun.apply("satya"));

System.***out***.println(lengthFun.apply("mnvv"));

}

**public** **static** **void** display(List<Resource> resources) {

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

**for** (Resource resource : resources) {

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

}

}

**public** **static** **void** display(List<Resource> resources, Condition condition) {

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

**for** (Resource resource : resources) {

**if** (condition.isOk(resource))

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

}

}

**public** **static** **void** displayBasedOnPredicate(List<Resource> resources, Predicate<Resource> predicate) {

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

**for** (Resource resource : resources) {

**if** (predicate.test(resource))

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

}

}

**public** **static** **void** displayBasedOnPredicate(List<Resource> resources, Predicate<Resource> predicate,

Consumer<Resource> consumer) {

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

**for** (Resource resource : resources) {

**if** (predicate.test(resource))

consumer.accept(resource);

}

}

}

**package** com.satya.java8ex.lambda;

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

/\*\*

\* Exception Handling in Lambdas

\*

\* **@author** SATYA

\*

\*/

**public** **class** LambdaTest7 {

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

*testCase2*();

}

**public** **static** **void** testCase1() {

**int**[] a = { 11, 21, 31, 41 };

**int** b = 0;

// process(someNumbers, key, (v,k) -> System.out.println(v/k));

*process*(a, b, (v, k) -> {

**try** {

System.***out***.println(v / k);

} **catch** (ArithmeticException e) {

System.***out***.println("ArithmeticException happend");

}

});

}

**public** **static** **void** testCase2() {

**int**[] a = { 11, 21, 31, 41 };

**int** b = 0;

// process(someNumbers, key, (v,k) -> System.out.println(v/k));

*process*(a, b, *wrapperLambda*((v, k) -> System.***out***.println(v / k)));

}

**private** **static** **void** process(**int**[] someNumbers, **int** key, BiConsumer<Integer, Integer> consumer) {

**for** (**int** i : someNumbers) {

consumer.accept(i, key);

}

}

**private** **static** BiConsumer<Integer, Integer> wrapperLambda(BiConsumer<Integer, Integer> consumer) {

**return** (v, k) -> {

**try** {

consumer.accept(v, k);

} **catch** (ArithmeticException e) {

System.***out***.println("Exception caught in wrapper lambda");

}

};

}

}

**package** com.satya.java8ex.lambda;

/\*\*

\* Closures in Lambda Expressions

\* **@author** SATYA

\*

\*/

**interface** Processor{

**public** **abstract** **void** process(**int** value);

}

**public** **class** LambdaTest8 {

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

**int** a=10;

**int** b=20; //final int b=20; must be in jdk1.7

*doProcess*(a,**new** Processor() {

@Override

**public** **void** process(**int** v) {

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

}

});

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

}

**public** **static** **void** doProcess(**int** i, Processor p) {

p.process(i);

}

}

**package** com.satya.java8ex.lambda;

**interface** Task {

**public** **abstract** **void** execute(**int** value);

}

**public** **class** LambdaTest9 {

**private** **int** someNonStaticData = 10;

**private** **static** **int** *someStaticData* = 11;

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

*doRunTask2*();

LambdaTest9 lambdaTest9 = **new** LambdaTest9();

System.***out***.println("lambdaTest9:" + lambdaTest9);

lambdaTest9.doRunTask1();

}

**public** **void** doRunTask1() {

System.***out***.println("\ninside doRuntTask this:" + **this**);

**int** p = 5;

Task task3 = (x) -> {

System.***out***.println(" non-static context lamdba expression");

System.***out***.println("lambda expression is accessing outer method local variable:" + p);

// p = 7;// c error because it is treated as final

System.***out***.println("lambda is processing value:" + x);

System.***out***.println("inside lamdba expression this:" + **this**); // not an error because this is inside non

// static context and this will refer to

// enclosing non static method object.

System.***out***.println("lambda expression is accessing data member:" + someNonStaticData);

System.***out***.println("lambda expression is accessing data member:" + **this**.someNonStaticData);

System.***out***.println("lambda is accessing someStaticData:" + *someStaticData*);

};

*doExecuteTask*(20, task3);

}

**public** **static** **void** doRunTask2() {

**int** a = 9;

Task task1 = **new** Task() {

@Override

**public** **void** execute(**int** v) {

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

// a=7;

System.***out***.println("anonymous innter class is processing the value:" + v);

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

}

};

Task task2 = (x) -> {

System.***out***.println("\n static context lamdba expression");

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

// a =11; c error it treats as final variable

System.***out***.println("lambda is processing value:" + x);

System.***out***.println("lambda is accessing someStaticData:" + *someStaticData*);

// System.out.println("this:"+this); // is error because this is static context

// System.out.println("lambda expression is accessing data

// member:"+someNonStaticData); c error

// System.out.println("lambda expression is accessing data

// member:"+this.someNonStaticData); c error

};

*doExecuteTask*(20, task1);

*doExecuteTask*(20, task2);

}

**public** **static** **void** doExecuteTask(**int** x, Task taskExecutor) {

taskExecutor.execute(x);

}

}

**package** com.satya.java8ex.lambda;

//Method References and collections

**public** **class** LambdaTest10 {

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

}

**public** **static** **void** testCase1() {

Thread t;

t = **new** Thread(() -> *printMsg*());

t.start();

}

**public** **static** **void** testCase2() {

Thread t;

t = **new** Thread(LambdaTest10::*printMsg*);

t.start();

}

**public** **static** **void** printMsg() {

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

}

}

/\*\*

\* The foreach iteraton and Java collection streams

\*/

**package** com.satya.java8ex.lambda;

**import** java.util.Arrays;

**import** java.util.List;

**class** Student {

**private** **int** id;

**private** String name;

**private** String branch;

**public** Student() {

}

**public** Student(**int** id, String name, String branch) {

**this**.id = id;

**this**.name = name;

**this**.branch = branch;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

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

**this**.name = name;

}

**public** String getBranch() {

**return** branch;

}

**public** **void** setBranch(String branch) {

**this**.branch = branch;

}

**public** String toString() {

**return** id + "," + name + "," + branch;

}

}

**public** **class** LambdaTest11 {

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

List<Student> students = Arrays.*asList*(**new** Student(11, "satya", "ece"),

**new** Student(12, "raju", "eee"),

**new** Student(13, "vinod", "cse"),

**new** Student(5, "yukthi", "civil"),

**new** Student(50, "ashok", "ece")

);

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

**for** (**int** i = 0; i < students.size(); i++)

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

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

**for** (Student s : students)

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

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

students.forEach(p -> System.***out***.println(p));

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

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

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

students.stream().forEach(p -> System.***out***.println(p));

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

**long** count = students.parallelStream().filter(p->p.getBranch().contains("ece")).count();

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

}

}

Anonymous inner class can extend concrete class.

Anonymous inner class can extend abstract class.

Anonymous inner class can implement interface with any number of methods.

But Lambda expression can not implement an interface with only single abstract method.

Lambda expression is a kind of method definition assigned to single abstract method defined in the functional interface.

Runner r = () -> { System.out.println("hi"); };

Hence if anonymous inner calss implements functional interface in that particular case we can replace anonymous inner class with lambda expression.

So we can conclude that anonymous inner class != lambda expression

Inside anonymous inner class we can delcare instance variables.

Inside anonymous inner class 'this' always refers current inner class object.

class A{

int a=9;

public void show(){

System.out.println(a);

}

}

class B{

int b=8;

void fun(){

A x = new A(){

int c=5;

public void show(){

System.out.println(a);

System.out.println(this.a);

System.out.println(c);

System.out.println(this.c);

}

};

x.show();

}

}

public class Demo{

public static void main(String[] args){

B b = new B();

b.fun();

}

}

Inside lambda expression can we declare instance variables?

ans: no

Lambda expression is not an implementation to interface.

Lambda expression is a kind of method definition to an abstract method declared in the functional interface.

So there is no question of instance variables.

Runnable r = ()->{

int c;

System.out.println("run method ");

};

What ever the variables we declared inside lambda expression are local variables.

Here we can use the 'this' reference inside lambda expression whenever the lambda expression is inside non static method.(here the 'this' will refer to outer class object)

we can not use the 'this' refence inside lambda expression whenever the lambda expression is inside static method.

and the

class A{

int a=5;

public void show(){

Runnable r = () -> {

System.out.println("a="+a);

System.out.println("this.a="+this.a);

};

Thread t = new Thread(r);

t.start();

}

}

public class Demo{

public static void main(String[] args){

A a1 = new A();

a1.show();

}

}

----

class A{

int a=5;

public static void show(){

Runnable r = () -> {

System.out.println("a="+a);// c error

System.out.println("this.a="+this.a);// c error

};

Thread t = new Thread(r);

t.start();

}

}

----

From Lambda expressions we can access enclosing class variables and enclsoing method variables directly.

The local variables ref from lambda expression are implicitly final and hence we can't perform re assignment for those local variables if we try to do so then we will get compilation error.

class A{

static int a=5;

public static void show(){

int x = 9;

Runnable r = () -> {

System.out.println("a="+a);// allowed

System.out.println("A.a="+A.a);// allowed

System.out.println("x="+x);

// x = 9; C error

};

Thread t = new Thread(r);

t.start();

}

}

|  |  |
| --- | --- |
| **Anonymous Inner class** | **Lambda Expression** |
| It is a class without name | It is a method without name (anonymous function) |
| Anonymous inner class can extend abstract and concrete classes | Lambda expression can’t extend abstract and concrete classes |
| Anonymous inner class can implement an interface that can contain any number of abstract methods | Lambda expression cannot implement an interface but it is used to provide method definition for a single abstract method declared in the interface. |
| Inside anonymous inner class we can declare instance variable | Inside lambda expression we can not declare instance variables, what eve variables declared are simply acts as local variables. |
| Anonymous inner class can be instantiated | Lambda expression cannot be instantiated. |
| Inside anonymous inner class “this always refers current anonymous inner class object but not outer class object | Inside lambda expression “this” always refers current outer class object. That is enclosing class object. (if the expression is belongs to object context) |
| Anonymous inner class is the best choice if we want to handle multiple methods. | Lambda expression is the best choice if we want to handle interface with single abstract method (functional interface) |
| In the case of anonymous inner class at the time of compilation a separate .class file will be generated | At the time of compilation no .clsss file will be generated for Lambda expression. It simply converts into to private method of outer class. |
| Memory allocated on demand whenever we are creating an object | Reside in permanent memory of JVM(method area) |

class Sample{

Runnable r = () -> {

System.out.println(“Hello”);

};

public void startIt(){

Thread t = new Thread( r ); t.start();

}

}

public class Demo{

public static void main(String[] args){

Sample s = new Sample(); s.startIt();

}

}