### Java 8

**Lambda Expression**

Lambda expression is **anonymous function** which **accepts a set of input parameters and return results**.

Lambda **enables functional programming for writing better code**.

**Lambda Expression is block of code without any name, with or without parameters and with or without results.**

**This block of code is executed on the demand.**

**optional type declaration, parentheses around parameter, curly braces, and return keyword are some of the characteristic of a lambda expression.**

Lambda Expressions **eliminates the need of anonymous classes**.

Lambda Expressions **can be used to implement functional interface**.

Lambda Expressions **are used primarily to define inline implementation of functional interface**.

Lambda function **have no access modifier**.

**A Lambda expression consists of two parts.**

The **Parameter part** and **expression part** separated by forward arrow.

Parameter -> expression

() -> System.out.println("HelloWorld");

This lambda expression doesn't have parameter and does return any results.

Lambda expression can be defined as an anonymous function that allows used to pass method as arguments. This help to remove boiler plate code.

**Functional Interface**

Functional Interface is **an interface which contains only one abstract method**.

Functional interface is also known as **Single Abstract Method** interface because it contains only one abstract method.

It **can have** **static method, default methods or can override object's class methods**.

As we have only one method in Runnable interface, hence it is considered as functional interface.

you can implement functional interface using lambda expression.

Runnable r = ( ()-> System.out.println("run method"));

Thread t = new Thread(r);

t.start();

**Functional Interface** – Runnable interface, Callable interface, Comparator Interface, Executor interface

Java can itself identify Functional Interface, but you can also specify interface as Functional interface by annotating it with @Functional interface.

**If you annotate @Functional interface, you should have only one abstract method otherwise you will get compilation error**.

@FunctionalInterface

**interface** Sayable {

**void** say(String message);

**static** **void** say1() { System.***out***.println("Test SAY1");}

}

**public** **class** TestFunctionalInterface {

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

Sayable sayable = (msg)->System.***out***.println("Test Message -->"+msg);

sayable.say("Functional Interface Example");

Sayable.*say1*();

}

}

**Output**:

Test Message -->Functional Interface Example

Test SAY1

|  |  |  |
| --- | --- | --- |
| **Consumer** | It takes one argument and return no result. **Accept**() | Consumer<T> c = c1.andThen(c2);  c.**accept**() |
| **Function** | It takes one argument and return a result. **Apply**() | Function<string> f = lambdaExpression;  f.**apply**() |
| **BiFunction** | It takes two argument and return a result. **Apply**() | Function<string,String> f = lambdaExpression;  f.**apply**() |
| **Predicate** | It takes one argument and return boolean. **Test**() | Predicate<String> p = str ->str.length() > 5  p.**test**(str); |
| **Supplier** | It takes no argument and return a result. **Get**() | Supplier<String> s = lambdaExpression;  s.**get**(); |
| **Binary operator** | similar to BiFunction. The two arguments, result is same type. |  |
| **unary operator** | similar to Function. The argument, result is same type. |  |

**Function Example**

**class** Student {

String name;

**int** marks;

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

**super**();

**this**.name = name;

**this**.marks = marks;

}

}

**public** **class** FunctionExample {

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

Function<Student, String> f = s -> {

**int** marks = s.marks;

String grade = "";

**if** (marks >= 80)

grade = "Distinction";

**else** **if** (marks >= 60)

grade = "First class";

**else**

grade = "Fail";

**return** grade;

};

Student st[] = { **new** Student("Kumaresh", 90), **new** Student("guru", 70) };

**for** (Student s2 : st) {

System.***out***.println("Name:" + s2.name);

System.***out***.println("Marks:" + s2.marks);

System.***out***.println("Grade:" + f.apply(s2));

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

}

}

}

**Output:**

Name:Kumaresh

Marks:90

Grade:Distinction

Name:guru

Marks:70

Grade:First class

**Consumer Example**

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

**class** Movie {

String name;

**public** Movie(String name) {

**super**();

**this**.name = name;

}

}

**public** **class** TestConsumerExample {

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

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

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

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

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

Movie t = **new** Movie("Guru");

cc.accept(t);

}

}

**Output:**

Guru Ready to Release

Guru Released

Guru Reviewed

**Test Supplier Example**

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

**public** **class** TestSupplierExample {

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

Supplier<String> s = ()->{

String otp ="";

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

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

}

**return** otp;

};

System.***out***.println("Six Digit OTP Number"+s.get());

}

}

**Output:**

Six Digit OTP Number887600

**Predicate Example**

**import** java.util.Arrays;

**import** java.util.List;

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

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

**import** java.util.stream.Collectors;

**public** **class** TestPredicateExample {

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

List<String> names = Arrays.*asList*("Adam","Alexandar","John","Kumaresh","Nikhilan","Kumaresan");

Predicate<String> predicate1 = str ->str.startsWith("K");

Predicate<String> predicate2 = str-> str.length() > 5;

List<String> filterList = names.stream().filter(predicate1.and(predicate2)).collect(Collectors.*toList*());

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

List<String> filterList2 = names.stream().filter(predicate2 ).collect(Collectors.*toList*());

System.***out***.println("filterList2: "+filterList2);

List<String> filterList3 = names.stream().filter(str->str.startsWith("K")).collect(Collectors.*toList*());

System.***out***.println("filterList3: "+filterList3);

filterList3.forEach(name->{ System.***out***.println("Name: "+name);});

BiPredicate<Integer,Integer> bi = (a,b)->a\*b>5;

System.***out***.println("BiPredicate Result-->"+bi.test(2,3));

System.***out***.println("BiPredicate Result-->"+bi.test(2,2));

}

}

**Output:**

filterList: [Kumaresh, Kumaresan]

filterList2: [Alexandar, Kumaresh, Nikhilan, Kumaresan]

filterList3: [Kumaresh, Kumaresan]

Name: Kumaresh

Name: Kumaresan

BiPredicate Result-->true

BiPredicate Result-->false

**Method Reference**

Method Reference **is used to refer method of the functional interface**.

It **is compact and easy form of lambda expression**.

A method reference **can be identified by** a **double colon**(::) **separating** a **class** or object **name** and **name of the method**.

**There are four kinds of method reference.**

1. Reference to static method (**Static method Reference**)

ContainingClass:StaticMethodName String::valueOf;

2. Reference to instance method of a particular object(**instance method reference**)

ContainingObject::instanceMethodName str::toString;

3. Reference to instance method of arbitrary object of particular type(instance method reference of arbitrary object )

ContainingType::methodName String::toString;

4. Reference to constructor (**Constructor Reference**)

classNAme :: new string::new

**ArbitraryObjectExample**

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.stream.Collectors;

**import** org.springframework.util.StringUtils;

**public** **class** ArbitraryObjectExample {

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

List<String> messages = Arrays.*asList*("hello", "baeldung", "readers!");

messages.forEach(word -> StringUtils.*capitalize*(word));

System.***out***.println("Lambda Expression :" + messages);

messages.forEach(**StringUtils::*capitalize***);

System.***out***.println("Method Reference :" + messages);

List<String> strings = Arrays.*asList*("how", "to", "do", "in", "java", "dot", "com");

List<String> sorted = strings.stream().sorted((s1, s2) -> s1.compareTo(s2)).collect(Collectors.*toList*());

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

List<String> sortedAlt = strings.stream().sorted(**String::compareTo**).collect(Collectors.*toList*());

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

}

}

**Output:**

Lambda Expression :[hello, baeldung, readers!]

Method Reference :[hello, baeldung, readers!]

[com, do, dot, how, in, java, to]

[com, do, dot, how, in, java, to]

**Constructor Reference Example**

**public** **class** ConstructorReferenceExample {

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

Messageable message = **Message::new**;

message.getMessage("Method Reference Constructor Testing");

}

}

**interface** Messageable {

Message getMessage(String message);

}

**class** Message {

Message(String message) {

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

}

}

**Output:**

Method Reference Constructor Testing

**instanceReferenceExample**

**interface** Sayable {

**public** **void** say();

}

**public** **class** instanceReferenceExample {

**public** **void** tetInterfaceMethod() {

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

}

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

instanceReferenceExample ir = **new** instanceReferenceExample();

Sayable sayable = **ir::tetInterfaceMethod**;

sayable.say();

Sayable sayb = new **instanceReferenceExample()::tetInterfaceMethod**;

sayb.say();

}

}

**Output:**

test interface method

test interface method

**StaticReferenceExample**

**interface** Sayable {

**public** **void** say();

}

**public** **class** StaticReferenceExample {

**public** **static** **void** tetInterfaceMethod() {

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

}

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

Sayable sayb = **StaticReferenceExample::*tetInterfaceMethod***;

sayb.say();

}

}

**Output:**

test interface method

|  |  |
| --- | --- |
| **Static method reference** | **interface** Sayable {  **public** **void** say();  }  **public** **class** StaticReferenceExample {  **public** **static** **void** tetInterfaceMethod() {  System.***out***.println("test interface method");  }  **public** **static** **void** main(String[] args) {  Sayable sayb = **StaticReferenceExample::*tetInterfaceMethod***;  sayb.say();  }  } |
| **Instance method reference of Particular Object** | **interface** Sayable {  **public** **void** say();  }  **public** **class** instanceReferenceExample {  **public** **void** tetInterfaceMethod() {  System.***out***.println("test interface method");  }  **public** **static** **void** main(String[] args) {  instanceReferenceExample ir = **new** instanceReferenceExample();  Sayable sayable = **ir::tetInterfaceMethod**;  sayable.say();  Sayable sayb = new **instanceReferenceExample()::tetInterfaceMethod**;  sayb.say();  }  } |
| **Instance method reference of arbitrary object** | List<String> messages = Arrays.*asList*("hello", "baeldung", "readers!");  messages.forEach(word -> StringUtils.*capitalize*(word));  System.***out***.println("Lambda Expression :" + messages);  messages.forEach(**StringUtils::*capitalize***);  System.***out***.println("Method Reference :" + messages); |
| **Constructor method reference** | **public** **class** ConstructorReferenceExample {  **public** **static** **void** main(String[] args) {  Messageable message = **Message::new**;  message.getMessage("Method Reference Constructor Testing");  }  }  **interface** Messageable {  Message getMessage(String message);  }  **class** Message {  Message(String message) {  System.***out***.println(message);  }  } |

**Stream**

A Stream is an iterator whose role is to accept a set of actions to apply on each of the elements it contains.

The Stream was designed to help improve collection processing and make it simpler in execution.

int sum = Arrays.Stream(new int[]{1,2,3})

.filter(i->i>=2)

.map(i->i\*3)

.sum();

**A Stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.**

**The Stream API is used to process collection of objects.**

**The feature of Java Streams are**

- A stream **takes input from** the collections, arrays or IO Channels.

- A stream **do not change the original data structure**; they **only provide the result as per the pipelined method**.

- **Each intermediate operation is lazily executed** and **returns a stream as a result**; **hence various intermediate operations can be pipelined**.

- **Terminal operations mark the end of the stream and return the result**.

- **Intermediate operations like filter, map, flatmap, sort return a stream and allows for further operation if required**.

- **Data can be processed with these operations only where there is a terminal operation like foreach, reduce, collect, sum,count etc.**

**These operations give a definite result, terminate the pipeline and start stream processing.**

**A Stream can be visualized as a pipeline**. **A stream pipeline consists of a source**(which might be an array, a collection), **zero or more intermediate operations**(which transforms a stream into another stream such as filter) **and a terminal operation**(which produces a result or side effect such as count() and forEach())

**Functional in nature**

An operation of stream produces a result but does not change the original data structure.

**Lazy Behaviour**

Intermediate operations are always lazy. **These operations do not start as soon as you reach that intermediate operation, it's only when stream hits the terminal operation that it starts executing operation**.

List<Integer> numList = Arrays.asList(34,6,3,12,65,1,8);

numList.stream().filter(n->n>5)

.sorted()

.forEach(System.out::println);

numList.stream().filter(n->n>5)

.sorted()

.collect(Collectors.toList());

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

**public** **class** StreamBuilderExample {

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

Stream.Builder<String> builder = Stream.*builder*();

builder.add("first")

.add("second")

.add("third");

Stream<String> stream = builder.build();

stream.forEach(System.***out***::print);

}

}

**Output :**

firstsecondthird

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.Comparator;

**import** java.util.Date;

**import** java.util.IntSummaryStatistics;

**import** java.util.List;

**import** java.util.Set;

**import** java.util.stream.Collectors;

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

**public** **class** CountStreamExample {

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

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

names.add("Ajeet");

names.add("Negan");

names.add("Aditya");

names.add("Steve");

names.add("Guru");

names.add("Kumaresh");

names.add("Guru");

**long** count = names.stream().filter(s->s.length()>=5).count();

System.***out***.println("Count: "+count);

List<String> sortedNames = names.stream().sorted().collect(Collectors.*toList*());

System.***out***.println("Sorted Names: "+sortedNames);

List<String> distinctNamest = names.stream().sorted().distinct().collect(Collectors.*toList*());

System.***out***.println("Distinct Names: "+distinctNamest);

names.stream().distinct().forEach(System.***out***::print);

List<Integer> n = Arrays.*asList*(2,3,4,5);

Stream<Integer> numberStream = n.stream();

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

List<Integer> number = Arrays.*asList*(2,3,4,5);

List<Integer> square = number.stream().map(x->x\*x).collect(Collectors.*toList*());

System.***out***.println("\nSquare List :"+square);

List<Integer> numberList = Arrays.*asList*(2,3,4,5);

Set<Integer> squareSet = numberList.stream().map(x->x\*x).collect(Collectors.*toSet*());

System.***out***.println("Square Set :"+square);

List<Integer> reduceList = Arrays.*asList*(2,3,4,2);

**int** reduceNumber = reduceList.stream().reduce(1, (x,y)-> x\*y);

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

List<String> strings = Arrays.*asList*("abc", "", "bc", "efg", "abcd","", "jkl");

**long** counts = strings.stream().filter(s->s.isEmpty()).count();

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

List<String> filteredString = strings.stream().filter(s->!s.isEmpty()).collect(Collectors.*toList*());

System.***out***.println("Actual String :"+strings);

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

List<String> limitString = strings.stream().limit(5).collect(Collectors.*toList*());

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

List<Integer> numbers = Arrays.*asList*(3, 2, 2, 3, 7, 3, 5);

IntSummaryStatistics stats = numbers.stream().mapToInt(x -> x).summaryStatistics();

System.***out***.println("Highest number in List : " + stats.getMax());

System.***out***.println("Lowest number in List : " + stats.getMin());

System.***out***.println("Sum of all numbers : " + stats.getSum());

System.***out***.println("Average of all numbers : " + stats.getAverage());

Stream<Date> stream = Stream.*generate*(() -> { **return** **new** Date(); }).limit(1);

stream.forEach(p -> System.***out***.println("Stream Generate Date :"+p));

List<String> stringList = Arrays.*asList*("abc", "zxx", "bc", "efg", "abcd","a", "jkl");

List<String> reverseStringList = stringList.stream().sorted(Comparator.*reverseOrder*()).collect(Collectors.*toList*());

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

}

}

**Output:**

Count: 5

Sorted Names: [Aditya, Ajeet, Guru, Guru, Kumaresh, Negan, Steve]

Distinct Names: [Aditya, Ajeet, Guru, Kumaresh, Negan, Steve]

AjeetNeganAdityaSteveGuruKumaresh2

3

4

5

Square List :[4, 9, 16, 25]

Square Set :[4, 9, 16, 25]

reduceNumber :48

counts :2

Actual String :[abc, , bc, efg, abcd, , jkl]

filteredString :[abc, bc, efg, abcd, jkl]

limitString :[abc, , bc, efg, abcd]

Highest number in List : 7

Lowest number in List : 2

Sum of all numbers : 25

Average of all numbers : 3.5714285714285716

Stream Generate Date :Sat Mar 14 17:39:55 IST 2020

reverseStringList :[zxx, jkl, efg, bc, abcd, abc, a]

**Optional**

Optional is **special wrapper class used for expressing optionally**.

optional is defined in java.util package.

**It wraps the single value if that value is available.**

**If the value is not available an empty optional should be returned.**

**optional class provides another way to handle situation when the value may or may not be present.**

**It is used to represent optional values that is either exist or not exist.**

It can contain either one value or zero value.

**If it contains a value, we can get it otherwise we get nothing.**

It is bounded collection that is it contains at most one element only. It is an alternative to null value.

Advantage

1. It is **used to avoid null checks**.

2. It is **used to avoid NullPointerException**.

String item = new Strign("test");

Optional<String> op = optional.of(item);

op.ifPresent(str->System.out.println(str.length()));

**import** java.util.Optional;

**public** **class** OptionalExample {

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

Optional<String> opt = Optional.*of*("MALE");

System.***out***.println("Non Empty Optional :"+opt);

System.***out***.println("Non Empty Optional -Gender value:"+opt.get());

System.***out***.println("Empty Optional :"+opt.*empty*());

System.***out***.println("isPresent Optional :"+opt.isPresent());

Optional<String> empty =Optional.*empty*();

//System.out.println("Empty Optional:"+empty.get());//Compile Time Error

System.***out***.println("Empty is Present :"+empty.isPresent());

opt.ifPresent(s->System.***out***.println("String Length :"+s.length()));

String nullName = **null**;

String name = Optional.*ofNullable*(nullName).orElse("Guru");

System.***out***.println("orElse Name :"+name);

name = Optional.*ofNullable*(nullName).orElseGet(()->"Gurunathan");

System.***out***.println("orElseGet Name :"+name);

//String nameException = Optional.ofNullable(nullName).orElseThrow(IllegalArgumentException::new);

//System.out.println("nameException :"+nameException);

String answer1 ="Yes";

String answer2= **null**;

String ans1 = Optional.*ofNullable*(answer1).get();

//String ans1 = Optional.ofNullable(answer1).orElse("YesElse");

System.***out***.println("ans1 :"+ ans1);

System.***out***.println("answer1 :"+Optional.*ofNullable*(answer1));

System.***out***.println("answer2 :"+Optional.*ofNullable*(answer2));

//NullPointerException

//System.out.println("answer2"+Optional.of(answer2));

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

}

}

**Output:**

Non Empty Optional :Optional[MALE]

Non Empty Optional -Gender value:MALE

Empty Optional :Optional.empty

isPresent Optional :true

Empty is Present :false

String Length :4

orElse Name :Guru

orElseGet Name :Gurunathan

ans1 :Yes

answer1 :Optional[Yes]

answer2 :Optional.empty

**static method in interface**

**Post Java 8 release, Static methods are allowed to define inside interface.**

This is new feature added in Java 8 which will act as a helper method

**static method in interface is an utility method or helper method which is an associated to an interface.** It is not associated to any object.

**We need static method because of following reasons**.

1. we can **keep helper or utility method specific to an interface** in the same interface **rather that in a separate utility class**.

2. **we don't need separate utility class** like collections, Arrays to keep utility method.

3. **Clear separation of responsibilities that is we do not need one utility class to keep all utility method of collection API.**

public interface MyInterface {

static int getDefaultMethod(){

return 0;

}

}

int num = myInterface.getDefaultAmount();

**public** **interface** Vehicle {

**static** **void** run() {

System.***out***.println("Interface static run method is called");

}

}

**public** **class** StaticMethodExample {

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

// Vehicle v = new Vehicle(); //interface can not be instantiated by new operator

Vehicle.*run*();

}

}

**Output:**

Interface static run method is called

**Default method in interface**

* **To define default method inside interface “*default*” keyword is must and it should prefix method declaration.**
* **An interface default method is defined in the same way a method will be defined in a class.**

- One difference is that interface, default method is preceded by the keyword default.

- **Default method is used to define a method with a default implementation. you can override the default method also to provide the more specific implementation for the method.**

- **Default method enable you to add new functionality to the interfaces of your libraries** and ensure binary compatibility with code written for older versions of those interfaces.

- There are 2 interfaces having same default methods and a class implements both this interfaces and results in ambiguity problem

* To resolve ambiguity problem in Java 8, override the conflicting method
* Now, if we want to invoke default method from any of the interfaces then call using super keyword
* For example, ***<interfaceName>.super.<defaultMethodName>***

//either override default method in A or B

@Override

**public** **void** defaultMethod() {

A.**super**.defaultMethod();

}

public interface Vehicle {

default string turnAlarmOn(){

system.out.println("Turning the vehicle alarm on.");

}

default string turnAlarmOff(){

System.out.println("Turning the vehicle alarm off.");

}

}

**forEach**

forEach method **is to iterate over a collection or stream and perform certain action on each element of it**.

**It is default method defined in Iterable interface**. Collection classes which extends Iterable interface can use forEach() loop for iteration.

List<Person> items = new ArrayList<>();

items.forEach(item -> System.out.println(item.getName());

hm.forEach((k,v) -> { System.out.println(""+k+"value"+v);};);

Based on who controls iterations, Java iterators can be classified into external and internal iterators.

**External Iterators:(for loop) (enhanced for loop)**

When using an external iterator, the client code performing the iteration controls the iteration.

The client **creates the iterator** and **instructs it when to advance to the next element**.

for(String name:names){

System.out.println(name);

}

**Internal Iterator (for-each loop)**

**When you use an internal iterator, it is the iterator itself controls the iteration.**

**The client code essentially says to iterator, "perform this operation on the elements in the collection".**

names.forEach(name->System.out.println(name));

The signature of the forEach method is this

default void forEach(Consumer<? Super T> action);

**stream iteration**

names.stream().forEach(System.out::println);

**import** java.util.ArrayList;

**import** java.util.HashMap;

**import** java.util.List;

**import** java.util.Map;

**public** **class** forEachExample {

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

Map<Integer, String> namesMap = **new** HashMap<>();

namesMap.put(1, "Larry");

namesMap.put(2, "Steve");

namesMap.put(3, "James");

namesMap.forEach((k,v)->System.***out***.print(" Key :"+k+" Value: "+v));

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

namesMap.entrySet().forEach(e->System.***out***.print(" Key :"+e.getKey()+" Value: "+e.getValue()));

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

al.add("Apple");

al.add("Lemon");

al.add("Bannana");

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

al.forEach(l->System.***out***.print(l+" "));

}

}

**Output:**

Key :1 Value: Larry Key :2 Value: Steve Key :3 Value: James

Key :1 Value: Larry Key :2 Value: Steve Key :3 Value: James

Apple Lemon Bannana