# Lambda Expressions and Functional Interfaces

An interface that only contains only a single abstract method then it is called a Functional Interface.

Lambda expressions are used to instantiate such functional interfaces , so the return type of Lamda expression is the object.

Anonymous class is only creating for one-time use, we cannot reuse it. The question that must be wondering that **why do we need such type of class?** Some scenarios, like when our only purpose is to override a method you can use it.

Let us say we have an add method below , but we could name it generically and it is possible to provide the implementation right away like subtract also.

**interface** **Operation**{

int add(int a , int b) ;

}

**class** **JavaMain** {

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

Io.initializeIO() ;

*//String name = Io.input.nextInt() ;*

*//Io.output.println( mapFreq ) ;*

Operation obj = (a , b )-> a+b ;

Io.output.println( obj.add(10 , 15 ) ) ;

Io.closeIO() ;

}

}

# Java – Lambda Expression Variable

*// Inrterface*

**interface** **MyFunction** {

*// Method inside the interface*

int func(int n);

}

*// Main class*

**class** **JavaMain** {

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

{

Io.initializeIO() ;

*// Custom local variable that can be captured*

int number = 10;

MyFunction myLambda = (n) ->

{

*// This use of number is OK It does not modify*

*// num*

int value = number + n;

*// However, the following is illegal because it*

*// attempts to modify the value of number*

*// number++;*

**return** value;

};

*// The following line would also cause an error,*

*// because it would remove the effectively final*

*// status from number.*

*//number = 9;*

int res = myLambda.func(20) ;

Io.output.println(res) ;

Io.closeIO() ;

}

}

# Java 8 | Consumer Interface

The Consumer Interface is a part of the **java.util.function** package.

It represents a function which takes in one argument and produces a result. However these kind of functions don’t return any value.

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

**class** **JavaMain** {

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

{

Io.initializeIO() ;

Consumer<Integer> display = a -> Io.output.println(a) ;

display.accept(20) ;

Io.closeIO() ;

}

}

# Java 8 | BiConsumer

It represents a function that takes in two arguments and doesn’t return any value.

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

**class** **JavaMain** {

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

{

Io.initializeIO() ;

BiConsumer<String, Integer> printEntry = (k , v ) -> Io.output.println("Key : " + k + ",Value : " + v ) ;

printEntry.accept("Seshrao" , 12 ) ;

Io.closeIO() ;

}

}

O/p :

Key : Seshrao,Value : 12

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

**class** **JavaMain** {

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

{

Io.initializeIO() ;

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

map.put("John", 30);

map.put("Jane", 25);

map.put("Bob", 40);

BiConsumer<String, Integer> printEntry = (k , v ) -> Io.output.println("Key : " + k + ",Value : " + v ) ;

map.forEach(printEntry) ;

Io.closeIO() ;

}

}

Key : Bob,Value : 40

Key : John,Value : 30

Key : Jane,Value : 25

# Java 8 Predicate

In Java 8, predicates are functional interfaces that allow you to define conditions or criteria to filter, test, or operate on data. Predicates are often used with streams to perform filtering operations on collections or arrays.

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

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

**class** **JavaMain** {

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

{

Io.initializeIO() ;

List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

Predicate<Integer> isEven = n -> n%2 == 0 ;

List<Integer> evenNums = nums.stream()

.filter(isEven)

.collect(Collectors.toList()) ;

Io.output.println(evenNums) ;

Io.closeIO() ;

}

}

O/p : [2, 4, 6, 8]

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

{

Io.initializeIO() ;

List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

Predicate<Integer> isEven = n -> n%2 == 0 ;

Predicate<Integer> isGreaterThan5 = n -> n>5 ;

Io.output.println(isEven.and(isGreaterThan5).test(12)) ;

List<Integer> resultList = nums.stream()

.filter(isEven.and(isGreaterThan5))

.collect(Collectors.toList()) ;

Io.output.println(resultList) ;

Io.closeIO() ;

}

O/p :

true

[6, 8]

{

Io.initializeIO() ;

List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

Predicate<Integer> isEven = n -> n%2 == 0 ;

Predicate<Integer> isGreaterThan5 = n -> n>5 ;

boolean res = isEven.and(isGreaterThan5.negate()).test(12) ;

Io.output.println(res) ;

Io.closeIO() ;

}

Output : false

# Supplier Interface

It represents a supplier of results, which means it doesn't take any input arguments but produces a result. It is often used when you need to generate or supply values lazily.

@FunctionalInterface

**public** **interface** **Supplier**<T> {

T get();

}

Lazy Initialization

Suppose you want to lazily initialize an object, for example, a database connection or a configuration object:

**class** **Person**{

**private** int id ;

**private** String name ;

**public** Person(){

**this**.id = 0 ;

**this**.name = "default name" ;

}

**public** Person(int id , String name ){

**this**.id = id ;

**this**.name = name ;

}

**public** String toString(){

**return** "{Id : "+**this**.id + " , Name : " + **this**.name + "}" ;

}

}

**class** **JavaMain** {

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

{

Io.initializeIO() ;

Supplier<Person> defaultPersonSupplier = () -> **new** Person() ;

Supplier<Person> personSupplier = () -> **new** Person(1 , "Person") ;

Io.output.println(defaultPersonSupplier.get()) ;

Io.output.println(personSupplier.get()) ;

Io.closeIO() ;

}

}

Generating Random Values

You can use a Supplier to generate random values on demand:

Supplier<Integer> randomIntegerSupplier = () -> **new** Random().nextInt(100);

int randomNumber1 = randomIntegerSupplier.get();

int randomNumber2 = randomIntegerSupplier.get();

Io.output.println("Random Number 1: " + randomNumber1);

Io.output.println("Random Number 2: " + randomNumber2);

# Function Interface

In Java 8, java.util.function.Function is a functional interface that represents a function that takes one argument of a specified type and produces a result of another type.

It has one abstract method called apply, which takes an argument of type T and returns a result of type R.

@FunctionalInterface

**public** **interface** **Function**<T, R> {

R apply(T t);

}

**class** **JavaMain** {

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

{

Io.initializeIO() ;

Function<String, Integer> stringLengthFunction = str -> str.length();

int length = stringLengthFunction.apply("Sesharao");

Io.output.println("Length of the string: " + length);

Io.output.println() ;

Io.closeIO() ;

}

}

O/p : 8

# Method References

Method references provide a concise way to refer to methods without executing them. They are often used in functional interfaces (interfaces with a single abstract method) like Function, Consumer, or Predicate.

It is a shorthand notation for a lambda expression that contains just one method call.

## Static Method Reference

Reference to a static method using the ClassName::staticMethodName syntax.

**class** **MethodRef** {

**public** **static** String convertToUpper(String str) {

**return** str.toUpperCase() ;

}

}

**class** **JavaMain** {

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

{

Io.initializeIO() ;

Function<String, String> convertToUpper = MethodRef::convertToUpper ;

convertToUpper.apply("Sesharao");

Io.output.println( convertToUpper.apply("Sesharao") );

Io.output.println() ;

Io.closeIO() ;

}

}

O/p : SESHARAO

## Instance Method Reference of a Particular Object

You can reference an instance method of an object using the instance::instanceMethodName syntax

**class** **MethodRef** {

**public** String convertToUpper(String str) {

**return** str.toUpperCase() ;

}

}

**class** **JavaMain** {

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

{

Io.initializeIO() ;

MethodRef methodRef = **new** MethodRef() ;

Function<String, String> convertToUpper = methodRef::convertToUpper ;

convertToUpper.apply("Sesharao");

Io.output.println( convertToUpper.apply("Sesharao") );

Io.output.println() ;

Io.closeIO() ;

}

}

## Instance Method Reference of an Arbitrary Object of a Particular Type

**class** **MethodRef** {

**public** void printIt(String message) {

System.out.println(message);

}

}

**class** **JavaMain** {

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

{

Io.initializeIO() ;

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

messages.add("Hello");

messages.add("Java");

messages.add("World");

MethodRef methodRef = **new** MethodRef() ;

Consumer<String> printIt = methodRef::printIt ;

printIt.accept("Sesharao");

messages.forEach(printIt) ;

Io.output.println() ;

Io.closeIO() ;

}

}

O/p :

Sesharao

Hello

Java

World

## Constructor Reference

You can reference a constructor using the ClassName::new syntax

**class** **MethodRef** {

**public** MethodRef(){

Io.output.println("MethodRef constructor is called") ;

}

}

**class** **JavaMain** {

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

{

Io.initializeIO() ;

Supplier<MethodRef> objectSupplier = MethodRef::**new** ;

MethodRef obj = objectSupplier.get() ;

Io.output.println() ;

Io.closeIO() ;

}

}

O/p : MethodRef constructor is called