# why java 8 was introduced ? what was the agenda behind java 8 ?

* Java 8 was introduced to bring conciseness in the code(reduce the boiler plate code , more readable and reusable)
* It brings functional programming which is enabled by Lambda Expressions ( powerful tool to create conside code base).
* Java lost a lot of market , due to Python and Scala can do the same thing
* with less LOC at ease , so to prevent further loss java upgraded itself and introduced the FP along with OOPs.
* Parallel operations can be done.

# Lambda Expressions and Functional Interfaces

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

It can have any number of static , default methods present in that.

Example : Runnable , Comparable

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

Whenever we have an existing implementation of Abstract method of our Functional interface , then we can go for the method reference (for code reusability).

So , method reference can be used as the replacement for the lambda expression.

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

# Default methods

It is a way for adding new methods to the interface without affecting the already/new implementing classes.

With this new feature , we can defend many compile time errors that may arise due to the unimplemeted methods of the interface.

**Is it ncessary to override the default methods ?**

Default methods have the dummy impementation , if the implementing class is needs to override , they can otherwise they can have the dummy implementation.

**Is default keyword one of the access modifiers ?**

The default keyword can not be used as an access modifier in Java unlike public / protected / private , because for default we do not use any keyword.

default keyword was only used in classes till java 1.8 for switch case alone , never in the interface.

Now it is used for default methods in interface to provide default implementation for all implementing classes to use.

So by default all the methods in the interface are public till java 1.8.

So default keyword will not act as a **access modifier** in interface.

**how the default methods in interface cope up with the diamond problem ?**

Java 8 has introduced a set of conflict resolution rules for inherited default methods in order of precedence

These rules help to resolve conflicts when inheriting default methods with the same signatures from multiple interfaces

The conflict resolution rules for inherited default methods in Java 8 are:

A method that is already implemented in the class or a superclass takes precedence over a default method.

If there is a conflict between two or more default methods, the class that implements the interface must provide its own implementation of the method.

# Static Methods in Interface

Why Static Methods were introduced in Interface in java 8 ?

Only reason for introducing static methods in interface is that you can call those methods with just interface name.

No need to create class and then create object.

Interface does not contain the static blocks / constuctors.

So , if you have everything static , then better go for interface rather than a class which is cost effective interms of memory.(you only have this flexibility after java 8 , before that you need to create class).

**Are the static methods in interface available to implementing classes by default ?**

They are not accessible with the reference of the implementing class , as they are static and the scope is only limited with the class/ interface itself.

However we can explicitly call those methods using the inteface name.

# Optional

The Optional operation in Java 8 is used to represent a value that may or may not be present. It is a container object that can hold both empty and non-null values. The purpose of the Optional class is to provide a type-level solution for dealing with null values.

It can be used to specify alternative values to return when something is null.

The Optional class can minimize the number of null checks in code by explicitly saying that a value can be null and setting proper default values.

Optional<String> optional = Optional.ofNullable(**null**);

**if** (optional.isPresent()) {

System.out.println(optional.get());

} **else** {

System.out.println("value is absent");

}

O/p : value is absent

Optional<String> optional = Optional.ofNullable("Sesh");

**if** (optional.isPresent()) {

System.out.println(optional.get());

} **else** {

System.out.println("value is absent");

}

O/p : Sesh

# PermGen and MetaSpace

In Java 8 and later versions, there has been a significant change in how memory is managed for class metadata and constants compared to earlier versions of Java.

Java 8 and later no longer use Permanent Generation (PermGen) for these purposes; instead, they use a new memory area called Metaspace.

PermGen (Permanent Generation) :

* In Java versions prior to Java 8, class metadata, such as class definitions and method metadata, and constants (String pool, etc.), were stored in a memory area known as the PermGen space (Permanent Generation). PermGen was a fixed-size memory area.
* One of the common issues with PermGen was that it could run out of space, especially in applications that dynamically generated or loaded many classes, leading to "OutOfMemoryError: PermGen space" errors.
* The PermGen space was part of the heap, and it was not garbage collected. It meant that if you had memory leaks in class loading or unloading, you could run into problems.

Metaspace

* In Java 8 and later, PermGen was replaced by a new memory area called Metaspace. Metaspace is part of the native memory (outside the heap) and is not subject to a fixed size like PermGen.
* Metaspace can automatically grow and shrink based on the application's requirements, making it more flexible and less prone to PermGen-related memory issues.
* Metaspace is garbage collected, meaning that if there is no longer any reference to a class or its metadata, the memory can be released. This helps to avoid memory leaks caused by class loading and unloading.
* You can configure the Metaspace size and other related parameters using command-line options when starting a Java application. This allows you to tune the Metaspace settings according to your application's needs.

Metaspace provides more flexibility and better memory management than PermGen, and it is an improvement in terms of stability and ease of use.