**Java 8**

**Features**:

1. **Lambda Expression**
2. **Functional Interfaces**
3. **Default methods in Interfaces**
4. **Static methods in Interface**
5. **Streams API**
6. **Predicate (Predefined Functional Interfaces)**
7. **Function (Predefined Functional Interfaces)**
8. **Consumer (Predefined Functional Interfaces)**
9. **Method reference & constructor reference by :: operator.**
10. **Date and Time API (introduced by joda.org, also known as Joda API)**
11. **Optional Class**
12. **JDK enhancement**

**Why Java 8**

* **By using java 8 we can write more concise, clean and maintainable code. (to simplify programming)**
* **By using java 8 we can enable functional programming in the form of Lambda expression**
* **To enable functional programming, java program can run on multi core processor**

**Functional Program eg:**

**p.s.v.m(){**

**Function<Integer,Integer> f =I->i\*I;**

**s.o.p(f.apply(2)) //4**

**//here instead of writing new method to get square of number we are using one line function to get square value**

**// Function<integer, integer> -- first param is input argument 2 nd param is output argument**

**}**

**1.Lambda Expression**

* **It is the anonymous function**
* **i.e is nameless function**
* **Anonymous function is the function without name, without return type, without modifiers**

**objective/advantage of lambda expression**

* **to enable functional programming in java**
* **To write more readable, maintainable, concise code**
* **To use API’s very easily and effectively**
* **To enable parallel processing**

**How to write lambda expression**

**Eg:1write simple hello world program**

**public void m1(){**

**s.o.p(“Hello world”);**

**}**

**Write lambda expression for above method**

**()-> {s.o.p(“Hello world”);}**

* **Removed modifier, method name, return type and added -> to indicate it is the lambda expression.**
* **Here {} contains method body**
* **If body has only one line, we can remove {}**
* **So, our lambda expression is () -> s.o.p(“Hello world”);**

**Eg: 2 : sum of two int numbers**

**(int a, int b)-> s.o.p(a+b);**

**Eg 3: square of int number**

**(int n)->{return n\*n};**

**(Or)**

**(int n)->n\*n**

**// without {} we should not use return , compiler will consider this is return n\*n**

**//within {} return statement is must**

**(n)->n\*n;**

**// int type removed, bcz compiler knows the type automatically based on value**

**n->n\*n // parenthesis removed bcz there is only one parameter.**

**Calling Lambda Expression**

* **To call lambda expression we need Functional interface**

**2. Functional Interface:**

* **Below are existing functional interfaces in Java8**

1. **Runnable -- run()**
2. **Comparable -- compareTo()**
3. **Comparator – compare()**
4. **ActionListener—actionPerformed()**
5. **Callable – call () - related to multithreading**

* **All the above interfaces contain single abstract method. Hence it is called Functional Interface**
* **In other words, an interface which contains SAM (Single Abstract Method), such type of interfaces is by default considered as functional interfaces**
* **Inside functional interfaces can contain multiple/any number of default methods and static methods also**
* **If we want to invoke/call lambda expression we need functional interface**

**Eg 1: functional Interface**

**@FunctionalInterface // it is annotation introduced in 1.8, to indicate compiler explicitly this interface is functional interface. Without this annotation also this interface considered as functional interface. Because it contains only one abstract method.**

**Interface inter{**

**public void m1();**

**default void m2(){**

**}**

**public static void m3(){**

**}**

**}**

**Eg 2:**

**@FunctionalInterface**

**interface A {**

**Public void m1();**

**}**

**@FunctionalInterface**

**interface B extends A {**

**}**

* **Here A class method is available for B interface through inheritance, hence child also contain only one abstract method so interface B also functional interface.**

**@FunctionalInterface**

**interface B extends A {**

**public void m1();**

**}**

* **It is also valid, bcz we have not introduced new abstract method it is already available in A interface.**
* **If we have introduced any new interface that time it become nonfunctional interface.**

**Eg 3: invalid functional interface**

**@FunctionalInterface**

**interface B extends A {**

**public void m2();**

**}**

* **Here interface B has two abstract methods m1,m2 and we used functional interface annotation hence it is invalid**

**Eg :4**

**interface B extends A {**

**public void m2();**

**}**

* **This valid compiler will not throw error**
* **Bcz here we are not using functional interface annotation. Hence it is considered as normal interface it can contain any number of abstract methods.**

**Eg 5:**

**interface A {**

**public void m1();**

**}**

**Class Demo implements A{**

**Public void m1()**

**{**

**s.o.p(“heloo...”);**

**}**

**}**

**class Test {**

**p.s.v.m(){**

**Demo d=new Demo();**

**d.m1();**

**}**

**Here demo class having m1 implementation using demo object we are calling m1 method**

**class Test {**

**p.s.v.m(){**

**A a=new Demo();**

**a.m1();**

**}**

* **Here A is parent interface. we can use parent reference to hold child object which is run time polymorphism.**
* **Instead of writing new class to provide implementation we can use lambda expression to provide implementation of function interface methods**

**Eg 6: write lambda expression provide implementation**

**interface A {**

**public void m1();**

**}**

**class Test {**

**p.s.v.m(){**

**A a= ()->s.o.p(“heloo....”); // provide implementation for m1 method using lambda expression**

**// here () --- is argument of m1 method**

**// {} -- implementation part of m1 method**

**a.m1(); // it will call interface A’s m1 method**

**}**

**Eg 6: write lambda expression provide implementation for add two numbers**

**interface A {**

**public void m1(int a, int b);**

**}**

**class Test {**

**p.s.v.m(){**

**A a= (a,b)->s.o.p(“sum(a+b): ”+(a+b)); // provide implementation for m1 method using lambda expression with 2 arguments**

**// here a is functional interface which contains only one abstract method m1.**

**// hence compiler will compare the m1 method in interface A and get the type of lambda expression automatically. So type in lambda expression arguments are optional.**

**a.m1(5,6); // it will call interface A’s m1 method**

**}**

**Note:**

* **Lambda expression always associate with functional interface only**
* **Wherever functional interfaces are there, there only we can use lambda expression**
* **There is no functional interface and no lambda expression**
* **If in interface A we have 2 methods m1,m2 with 2 int arguments and we are using lambda expression, how compiler will decide whether these lambda arguments are to m1 method or m2 method. To resolve these kinds of problem our functional interface always should contain only one abstract method**
* **Hence lambda expression is very specific concept, it is applicable to functional interfaces only**

**Difference B/w Anonymous inner class and lambda expressions**

|  |  |  |
| --- | --- | --- |
| **S.No** | **Anonymous inner class** | **lambda expressions** |
| **1** | **It is a class without name** | **It is a method name (Anonymous function)** |
| **2** | **It can extend abstract, concreate class** | **It can’t extend abstract, concreate class. It can implement only functional interface** |
| **3** | **It can implement interface which contains any number of abstract methods** | **It can implement interface which contains only one abstract method (Functional Interface)** |
| **4** | **Inside anonymous class we can create instance variable** | **Inside lambda expression we can’t declare instance variable, whatever variable declared are considered as local variable** |
| **5** | **Anonymous inner class can be instantiated** | **It can’t be instantiated** |
| **6** | **Inside anonymous inner class, this always refers current anonymous inner class object not outer class object** | **Inside lambda expression this always refers current outer class object, I.e enclosing user class** |
| **7** | **Anonymous inner class is best choice if want to handle Interface with multiple methods** | **It is best choice if want to handle Interface with single method (FI)** |
| **8** | **At the time compilation separate .class file created** | **Here separate .class file not created** |
| **9** | **Memory will be created in heap area on demand whenever object created** | **Lambda expression will reside in permanent memory of JVM (method area)** |
|  |  |  |
|  |  |  |

**3. Default Method()**

**Until 1.7v:**

* **Every method present inside the interface is always public and abstract even we have not declared**
* **Below methods are same in interface**

1. **Void m1();**
2. **Public void m1();**
3. **Abstract void m1();**
4. **Public abstract void m1();**

**1.8 V:**

**Default methods and static methods allowed**

**1.9 V:**

**Private methods allowed**

**Variables**

**In all version variables are public static final**

**Default Methods:**

* **It is also known as virtual extension method or defender method (protector method)**
* **If we defined the interface with 2 abstract methods m1, m2 all the implementation class should provide the implementation for the methods**
* **If we added any new method in interface all the implementation class will get impact**
* **To resolve this problem, we r using default method**
* **Without affecting the implemented classes to add the new method in interface we can go for default method**
* **This default methods provide common implementation for all the child classes if required we can override default method in child class**

**Eg :**

**interface A{**

**void m1();**

**void m2();**

**default void m3(){**

**s.o.p(“default implementation=====”);**

**}**

**public class Test implements A{**

**public void m3(){**

**s.o.p(“overriding default implementation=====”);**

**}**

**p.s.v.m(){**

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

**t.m3() // overriding default implementation=====**

**}**

**}**

**// default keyword is required to indicate this is default method.**

**//this m3 method implementation is common for all the child class and can override if current implementation not satisfied**

**}**

* **If we are overriding default method in child classes, we have to use public modifier only not default**
* **Default methods applicable only in interface not in classes**
* **Object class methods we can’t implement as default methods.**
* **Bcz object class method by default available to all the classes.**
* **all java classes direct or indirect child of object class**
* **Eg: hashCode(), equals (), tostring(),**
* **Simple we are not allowing to override object class methods as default methods.**
* **In java multiple inheritance not possible through class, but we can achieve through interface (method body available in child class only hence no ambiguity problem)**
* **But in default method if 2 interfaces left and right has same default method m1 in child class if you called m1 method which parent get consider**

**Interface left{**

**Default void m1()**

**{**

**s.o.p(“left interface m1 method====);**

**}**

**}**

**Interface right {**

**Default void m1() {**

**s.o.p(“right interface m1 method========”);**

**}**

**}**

**Class Test implements Left, Right{**

**p.s.v.m(){**

**s.o.p(“main method);**

**}**

**}**

* **This will throw the exception of ambiguity problem,**
* **To override this problem, we have to override default method in child class like below**

**Class Test implements Left , Right{**

**Public void m1(){**

**s.o.p(“our own m1 method===);**

**}**

**p.s.v.m(){**

**s.o.p(“main method);**

**}**

**}**

* **If we want our left default method**

**Class Test implements Left, Right{**

**Public void m1(){**

**Left.super.m1();**

**// similarly, we can call Right.super.m1();**

**}**

**p.s.v.m(){**

**s.o.p(“main method);**

**}**

**}**

**4.Static method ()**

* **We can call instance method by using class object only**
* **Static method called using class name.**
* **Interface is the light weight component, not that much costly bcz we never going to create object for interface and class is the heavy weight component and it is costly.**
* **If we have only static method no concreate method means then why we have to create class and we can go for interface itself**
* **Interface static method by default not available to implementation classes**

**Eg:**

**interface A {**

**Public static void m1() {**

**s.o.p(“default implementation=====”);**

**}**

**public class Test implements A {**

**p.s.v.m(){**

**//different way to call interface static method**

**// call using interface name**

**A.m1();**

**//this class Test is child class of interface A, but still interface static method by default not available to child class Test. So below are invalid call**

**m1();**

**Test.m1();**

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

**t.m1();**

**}**

**}**

* **Interface static method should call by using interface name only, interface\_name.method\_name(); and should not call using implementation class name**
* **We can have main method also inside interface , hence we can run the interface directly in cmd prompt.**
* **We can use the interface static method to define general utility methods.**

**5.Stream API**

* **If you want to represent a group of objects as single entity then we should go for collection**
* **If we want to process objects from the collection then we should go for streams concept**
* **IO stream is always talk about stream of data , collection stream always talks about stream of data.**
* **If we want to get the stream for collection c use below method**

**Stram s = c.stream();**

* **Stream is a interface present inside the java.util.stream package**
* **If we want to filter elements from collection object based on some conditions** **then we can use filter ()**

**Eg: filter even numbers and convert to list**

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

**l.add(0);**

**l.add(5);**

**l.add(10);**

**l.add(15);**

**l.add(20);**

**l.add(25);**

**s.o.p(l);//0,5,10,15,20,25**

**List<Integer> l2 = l.stream().filter(I->I%2==0).collect(Collectors.toList());**

**s.o.p(l2);//0,10,20**

**l.stream() -- will give you the stream object.**

**filter(I->I%2==0) -- will iterate the stream object and filters what are the numbers are integer**

**Collect(Collectors.toList()) -- collecting the filter output and converting to list.**

* **For every object if you want to perform some function and we want some new(result) object then go for map**

**List<Integer> l2 = l.stream().map(I->I+5).collect(Collectors.toList()));**

**s.o,.p(l2); //5,10,15,20,25,30**

* **Filter (Predicate)**
* **Map (Function)**
* **Filter – it is used to check Boolean condition return the same values**
* **Map – it is used to do some operation and return new values**
* **In filter if input is 10 elements, output may 0 to 10.**
* **In map if input is 10 elements, then output is 10 elements.**

**Long noOfFailedStudent = l.stream().filter(I->I<15).count();**

* **Count() -- will give u the count of integer less than 15.**
* **s.o.p(noOfFailedStudent) //3**

**List<Integer> l2 = l.stream().sorted().collect(Collectors.toList())**

* **will give the arraylist values in ascending order**
* **sorted () -- used to sort the stream of objects in ascending order**
* **if we are not passing any arg in sorted(), then internally i1.compareTo(i2) will called**

**Comparator --> compare(o1,o2)**

* **Return –1 , iff o1 < o2, o1 come first**
* **Return +1,iff o1 >o2 , o2 come first**
* **Return 0 iff o1 & o2 are equal.**

**List<Integer> l2 = l.stream().sorted((i1,i2)->((i1<i2)?1:(i1>i2)?-1:0))).collect(Collectors.toList())**

* **sorted (Comparator) -- used to customized sorting**
* **Since Comparator is functional interface, we can replace with lambda expression**
* **(i1,i2)->((i1<i2)?1:(i1>i2)?-1:0)) -- lambda expression to provide the implementation of comparator compare() method.**
* **we can rewrite this method like below**

**(i1,i2)->-i1.compareTo(i2)**

**or**

**(i1,i2)->i2.compareTo(i1)**

**Integer min = l.stream().min((i1,i2)->-i1.compareTo(i2)).get();**

**s.o.p(min)//0**

**Integer max = l.stream().max((i1,i2)->-i1.compareTo(i2)).get();**

**s.o.p(max)//30**

* **Min() - used to get the min value**
* **Max() - to get max value**
* **Get() -to return the value**

**l.stream().forEach(function)**

**l.stream().forEach(System.out::println)**

* **ForEach always takes function here**
* **Function is functional interface**
* **Hence, we can replace with lambda expression**
* **Wherever lambda expression is there we can replace with :: also**

**Consumer<Integer,void> f = I->System.out.println(“the square of “+i+” is :i\*i);**

**l.stream().forEach(f)**

* **We can call our own function here**

**Integer [] I = l.stream().toArray(Integer[]::new)**

* **toArray() - convert the arraylist to array**
* **While converting array we need to pass integer array object**
* **Integer[]::new – constructor reference which give you the integer[] object.**

**Stream.of(i).forEach(System.out::println);**

* **Stream.of(i) - to get the stream for Array.**
* **I – it tis array object**

1. **Stream() -- to create a stream**
2. **Filter() - to configure filter**
3. **Map()- to configure map**
4. **Collect() - to collect values after filter/map**
5. **Count()- to get the number of elements**
6. **Sorted() - to get sorting**
7. **Sorted(Comparator) - to customize sorting**
8. **Min(Comparator) - min element according to comparator**
9. **Max(Comparator) - max element according to comparator**
10. **ForEach() -- apply forEach iteration**
11. **ToArray() -- convert to Array—collection already contains toArray so no need of this.**
12. **Stream.of(array); -- to get stream object**
13. **MapToInt()**
14. **Sum()**
15. **Peak()- it is intermediate function , mainly used for debugging purpose**

**Note : https://javaconceptoftheday.com/solving-real-time-queries-using-java-8-features-employee-management-system/**

**6.Predicate**

* **It is predefined/build in functional interface**
* **It used to perform Boolean/conditional check**
* **It is part of java.utill.function package**
* **It has abstract method called test**
* **Which accepts one argument and returns Boolean value**
* **Passed as an argument in Filter method of Stream api , to perform filter operation using conditional check**

Predicate<Employee> salary = e -> e.getSalary()>20000

List<Employee> maleEmployeeList = employeeList.stream().filter(salary).toList();

**Eg1;**

**p.s.v.m()**

**{**

**Predicate<Employee> p1= e->e.getAge<30 && e.getSalary>20000;**

**s.o.p(p1.test(e));**

**}**

**Eg 2: simple program to find the length of string print even length strings**

**p.s.v.m()**

**{**

**String[] str= {“Nag”,”chiranjeev”,”Venkatesh”,”Balesh”.”Suresh”,”Kartina”}**

**Predicate<Employee> p= s->s.length()%2==0;**

**For(String s: str){**

**If(p.test(s))**

**s.o.p(s);**

**}**

**}**

**Eg 3: write program to combine 2 predicate**

**p.s.v.m(){**

**Int[] x= {0,5,10,15,20,25,30,35};**

**Predicate<Integer> p1= I->I%2 == 0;**

**Predicate<Integer> p2= I->I>10;**

**s.o.p(“the numbers which are even and > 10 are:”);**

**For(int x1 : x){**

**If(p1.and(p2).test(x1)) {**

**s.o.p(x1);**

**}}**

**s.o.p(“the numbers which are even or > 10 are:”);**

**For(int x1 : x){**

**If(p1.or(p2).test(x1)){**

**s.o.p(x1);**

**}}**

**s.o.p(“the numbers which are not even” );**

**For(int x1 : x){**

**If(p1.negate().test(x1)){**

**s.o.p(x1);**

**}}**

**}**

**Here we are combining predicates p1, p2**

**7. Consumer**

* **It is Functional Interface part of java.utill.function package**
* **Used to enable functional programming in java**
* **It has accept(Object) abstract method which will accept single input , perform some operation and returns no output,**
* **Consumer Interface passed as argument in forEach() method**

**Eg:** Consumer<String> printConsumer = name -> System.out.println(name);

Names.forEach(printConsumer);

**7.Function**

* **Functions also a predefined / built in functional interface**
* **It is part of java.utill.function package**
* **Function has abstract method called apply(Object), which take some input, perform some operations and returns some results**
* **Function passed as argument in Stream API ‘s Map method.**

**Eg:**  Function<Integer, Double> half = a -> a / 2.0;

        // Applying the function to get the result

        System.out.println(half.apply(10));

**9. Optional**

**10.Constructor reference**

**11.PremGem & MetaSpace**

**12.JDK enhancements**

**13.hashmap enhancements**

**14. Date and time API**

**Interview Questions**

**1.find the vowels count in the string using streams**

**String input=”hello hello”;**

**Long count = Input.chars().filter((x)->{**

**return (x==’a’|| x==’i’||x==’o’||x==’e’||x==’u’);**

**}).count();**

**Here chars() -** **with which we can obtain an instance of Stream from a String object. This simple API returns an instance of IntStream from the input String.**

**Simply put, IntStream contains an integer representation of the characters from the String object:**

**Map<String,Long> out = s.chars().mapToObj(x->(char)x).collect(Collectors.*groupingBy*(Object::toString, Collectors.*counting*()));**

**out.forEach((key,value)->System.*out*.println(key+” – “+value));**

* **Here s.chars – provide intstream**
* **mapToObj() – mapping each integer representation to equivalent char through type casting(down casting)**
* **grounpingBy() – we are grouping each char and counting the each char**

**2. what enhancements have been done to JDK**

* **until java 7, JVM used an area called PermGen to store Classes it got removed in java 8 and replaced by metaspace**
* **Major advantage of metaspace over permgen :**

**Permgen was fixed in term of maximum size and can not grow dynamically but metaspace can grow dynamically and don’t have any size constraints.**

**3. what enhancement have been done to Hashmap in java 8**

* **A bucket is one element of HashMap array. It is used to store nodes.**
* **Until java 7 , Two or more nodes can have the same bucket. In that case link list structure is used to connect the nodes.**
* **In java 8 it is replaced with balanced tree instead of linked list**
* **While converting the list to binary tree, hashcode is used as a branching variable. If there are two different hashcodes in the same bucket, one is considered bigger and goes to the right of the tree and other one to the left. But when both the hashcodes are equal, HashMap assumes that the keys are comparable, and compares the key to determine the direction so that some order can be maintained.**

**Reference:** [**https://www.nagarro.com/en/blog/post/24/performance-improvement-for-hashmap-in-java-8**](https://www.nagarro.com/en/blog/post/24/performance-improvement-for-hashmap-in-java-8)

**4.how to remove duplicates from list**

**1. In set duplicates are not allowed , if we add the list in to set duplicates get removed**

**Integer[] a = new Integer[] {1,2,3,4,1,2,4,3};**

**List<Integer> l = Arrays.*asList*(a);**

**Set<Integer> s = new HashSet<>(l);**

**2. using stream we can convert the list into Stream and collect the data into Set, hence duplicates removed**

**Set<Integer> s1 = l.stream().collect(Collectors.*toSet*());**

1. **How lambda expression and functional interfaces are related**

* **Lambda expressions can only be applied to functional interface abstract method**
* **If there is no functional interface there is no chance of using lambda expression**

**6.what is stream pipelining**

**7.what are functional interfaces**

* **Functional interface is a interface which contain only one abstract method and any number of default and static method**
* **These interfaces by default consider as functional interface**
* **By using @functionalInterface annotation also we can make the interface as functional interface , this annotation tells to compiler that this interface is functional interface, so if we add additional abstract method compiler will throw error.**
* **Many FI already present in java such as comparable, comparator, Runnable**

**8.method reference in java**

**9.OptionalClass**

**10. why default method needed in the interface**

* **Default methods let you add new functionality to your Interfaces and ensure binary compatibility with older code written for the interfaces.**

**11. findFirst() Vs findAny()**

* **findFirst() – returns the first element meeting the criterion, while findAny() returns any element meeting the stream, a feature that is very useful when working with a parallel stream.**
* **Eg: findFirst()**

**List<String> list = Arrays.asList("A", "B", "C", "D"); Optional<String> result = list.stream().findFirst();**

* **Here if the stream is empty findFirst() returns empty, hence return type is optional instance**
* **Eg: findAny()**

**Optional<Integer> list = l.stream().findAny();**

**System.*out*.println("====set values ===s1"+list);//any random element**

* **Here if the stream is empty findAny() returns empty, hence return type is optional instance**

**7.Function**

* **It is predefined /built in functional interface**
* **It is used to perform some operation and produce new result**
* **Result may be any type.**