Mostly Asked Java 8 Interview Questions

1) Why JAVA 8? Main agenda behind Java 8.

→ Java 8 was to introduce Conciseness in the Code.

→ Java brings in functional programming which is enabled by Lambda expressions (a powerful tool to create Concise code base).

→ If we have ever observed, with python, Scala we can do the same thing in very less LOC (lines of code). By mid 20s Java lost a large market due to these languages. To prevent further loss java upgraded itself from only OOPS language to some concepts of FP to create Concise code base.

2) What are the new features which got introduced in Java 8?

→ Lambda Expressions
→ Stream API
→ Default methods in the interfaces
→ Static methods
→ functional Interfaces
→ Optional
→ Method references
→ Date API
→ Nashorn, Javascript Engine.

3) What are main advantages of using Java 8?

→ Compact code (less boiler plate code)
→ More readable and reusable code
→ More testable code
→ parallel operations are possible.

4) What is lambda expression?

→ Lambda expression is an anonymous function (without name,

Eg: Normal programming technique

```
public void add (int a, int b){
    System.out.println(a+b);
}
```

Equivalent Lambda Expression

```
(a,b) -> System.out.println(a+b);
```

```
BiConsumer<Integer, Integer> biConsumer = (a,b) -> sop(a+b);
biConsumer.accept(10,5);
```

---

1) What are functional interfaces?

→ Functional interfaces are those interfaces which can have only one abstract method.

→ It can have any no. of static methods, default methods. No restrictions on that.

→ There are many functional interfaces already present in java such as eg: Comparable, Runnable

---

2) How Lambda expression and functional interfaces are related?

Functional Interface is used to provide reference to lambda expression. → This is the relation.

```
Comparator<String> c = (s1, s2) -> s1.compareTo(s2);
(s1, s2) -> s1.compareTo(s2) ; This is lambda Expression
Comparator<String> c  : This is functional Interface.
```

---

3) Can you create your own functional interface?

→ As we know functional interface is an interface with Exactly one Single Abstract method and can have multiple static or default methods.

→ To create our own functional interface:

* Create an Interface
* with @functionalInterface
* method.

Security that in case if u by mistake add 2 abstract methods then compiler will throw compile time error.

Eg:-
```
@FunctionalInterface
public interface FunctionalInterfaceDemo {

    void SingleAbstMethod();

}
```

---

8) What are Streams?

→ If we want to process bulk objects of collection then go for Streams Concept.

→ Way to operate on Collection in java 8 is Stream.

→ It a special iterator class that allows processing collections of objects in a functional manner.

Eg:- fetch all objects from collection of list whose value is greater than 15

```
List<Integer> arlist = new ArrayList<Integer>();
arlist.add(15);
arlist.add(25);
arlist.add(5);

List<Integer> newAL = new ArrayList<Integer>();
newAL = arlist.stream().filter(x -> x>=15).collect(Collectors.toList());
newAL.stream().forEach(x -> System.out.println(x));
```

---

9) Difference between Streams (java 1.8) and java.io.Stream?

→ java io streams is a sequence of characters or binary data which is used to be written to a file or to read data from a file.

→ java io streams related to file whereas java 8 streams are related to Collection object.

→ Hence if we need to perform some operations on collection then

11) Steps to create and process Stream.

Stream s = CollectionObject.stream();

Once we get Stream object we can process the object of Collection.

Processing of Stream consists of 2 steps/stages.

* Configuration of Stream
* Processing that configuration.

Configuration can be done by

* Map
* filter.

12) How to filter the Stream objects.

Stream s = collectionobject.stream().filter(i -> i%2==0)

Eg: List<Integer> arlist = new ArrayList<Integer>();

arlist.add(15);
arlist.add(25);
arlist.add(52);

Stream s = arlist.stream().filter(i -> i%2==0);
s.forEach(x -> System.out.println(x));

13) How to Map the Stream objects.

→ what if we don't want to filter out,
→ we rather want to create new object against each existing stream object based on some function.
→ EG in given stream create new object by squaring its value.

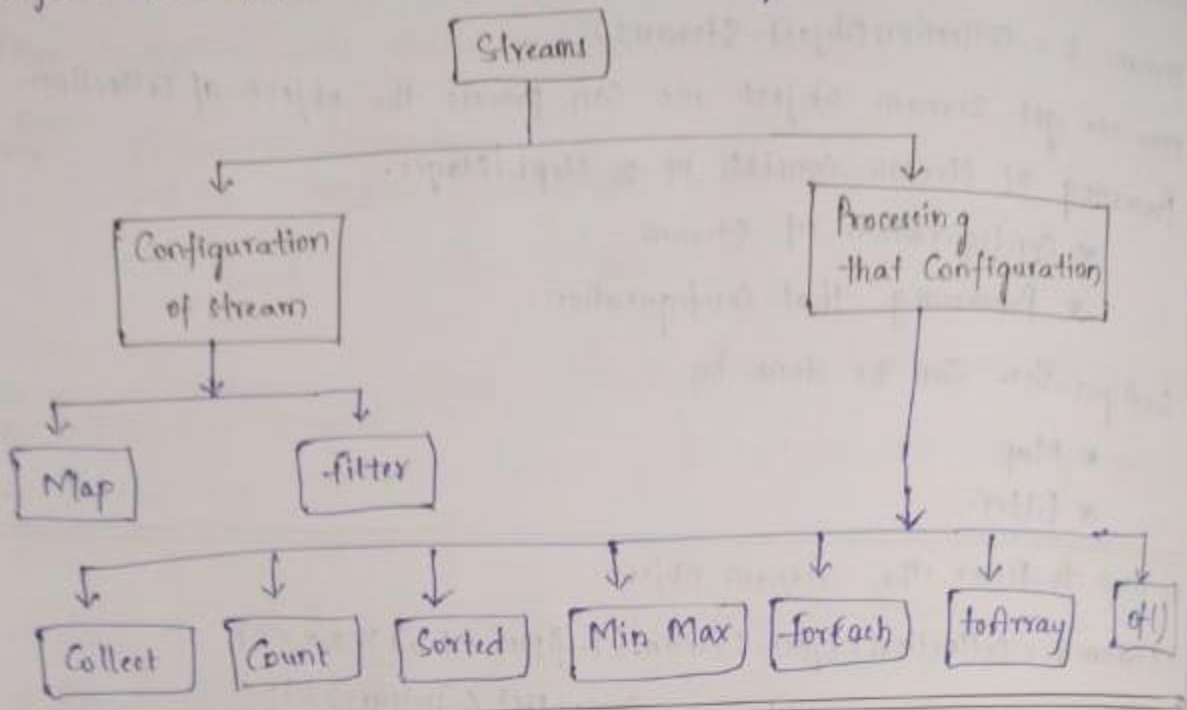
Stream s = arlist.stream().map(i -> i*i);
s.forEach(x -> System.out.println(x));

14) Difference between filter and Map.

→ If we want to fetch/filter objects from collection like e.g.: filter only even numbers from array list collection then use filter for configuration of Stream.

→ If we want to perform some operation on each objects of the

objects are there in both new and original list created.



15) How to process elements using collect()?

If we want to collect elements of stream after filtering or mapping and add them to the required Collection then use collect method.

```
List<Integer> newfilteredlist = arlist.stream().filter(i->i>=20).
                                collect(Collectors.toList());

newfilteredlist.forEach(x->System.out.println(x));
```

16) How to process elements using count()?

If we want to count how many elements are there in Collection that satisfy given condition then use count method.

```
long noffilteredlistCount = arlist.stream().filter(i->i>=20).count();
```

17) How to process elements using sorted()?

→ If we want to sort elements inside a stream use this sorted() method.

→ We can sort based on default natural sorting order

→ If we want to sort using customized sorting order then use Comparator.

```
Stream<Integer> newfilteredSortedlist = arlist.stream().sorted()
```

18) How to process elements using sorted in Descending order?

```
Stream<Integer> newfilteredSortedlist = arlist.stream().filter
                (i -> i>=20).sorted((i1, i2) -> i2.compareTo(i1));

newfilteredSortedlist.forEach(x -> System.out.println(x));
```

---

19) How to process elements using Min, Max?

→ Min(Comparator) will return the minimum value based on the defined comparator.

→ Max(Comparator) will return the maximum value based on the defined Comparator.

---

20)
```
Integer minvalue = arlist.stream().min((i1, i2) -> i1.compareTo(i2)).
                                                    get();
SOP(minvalue);
Integer maxvalue = arlist.stream().max((i1, i2) -> i1.compareTo(i2)).
                                                    get();
SOP(maxvalue);
```

---

21) How to Process elements using forEach?

→ forEach() is a method.

→ All methods that we saw till now returned something, like min max value, Sorted collections, etc

→ This method does not return anything.

→ Rather This method takes lambda expression as argument and apply that lambda expression to each element present in that Stream.

---

22) How to process elements using toArray()?

We can use this method to copy elements present in the Stream to specified array.

```
Object[] intArrOneLiner = arlist.stream().filter(i -> i>=20).
                                                    toArray();
for(Object o: intArrOneliner) {
    ... in Array is" + o);
```

22) How to process elements using of()?

→ Stream concept is not applicable just for the collections it's also applicable for "ANY GROUP of VALUE".

→ Even for arrays you can use stream.

→ Stream.of() this method can take any group of values and convert them to stream so that multiple stream operations can be applied to it.

```
Stream.of(1, 11, 111, 1111).forEach(x -> System.out.println(x));

String[] names = {"Code", "Decode", "Demo"};

Stream.of(names).filter(x -> x.length() > 4).forEach(x -> sop(x));
```

23) What is a Parallel Stream?

→ Java parallel Streams came into picture after java 1.8.

→ Its meant to utilize cores of processor.

→ Till Now our java code has 1 Stream of processing where it executes sequentially

→ But when you use parallel Streams, we divide code into multiple Streams that executes parallely, on separate cores and final result is the outcome of individual cores outcomes combined.

Tasks	Core	Task1	Task2	Task3	Task4
T1	Core1	T1	T2	T3	T4
T2	Core2				
T3	Core3				
T4	Core4				

→ The output of this sequential stream is T1, T2, T3, T4 → in sequential order tasks are executed and output of 1 can be input to another

Tasks	Core	Task1	Task2	Task3	Task4
T1	Core1		T2		
T2	Core2				T4
T3	Core3	T1			
T4	Core4			T3	

→ The output of this parallel stream is T2, T4, T1, T3 → not in sequential order.

→ Order of execution is not under control

→ Hence it is advisable to use parallel stream only when order of execution of threads does not matter and state of one element does not affect another.

---

14) What is an Intermediate Operation?

The operation which return another stream as a result are called intermediate operations. Very important part is they are lazy.

ex:- filter(), map(), distinct(), sorted(), limit(), skip()

15) What is Terminal Operation?

The operations which returns non-stream values like primitives or object or collection or return nothing are called terminal operations.

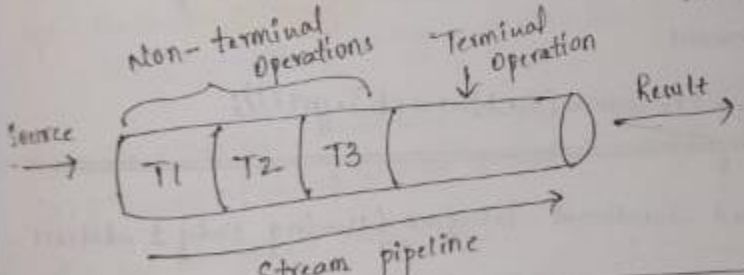
You can chain multiple intermediate operations and none of them will do anything until you invoke a terminal operation. At that time, all of the intermediate operations that you invoked earlier will be invoked along with the terminal operation.

ex:- forEach(), toArray(), reduce(), collect(), min(), max(), count(), anyMatch(), allMatch(), noneMatch(), findFirst(), findAny()

eg:- List<Integer> intList = Arrays.asList(1,2,3,4,5);

sop(intList.stream().filter(a->a%2==0).map(a->a+a).filter(a->a>6).count());

T1 T2 T3 Terminal operation.



---

16) Terminal Vs Intermediate Operations.

Intermediate Operations	Terminal Operations
→ They return Stream	→ They return non-stream values.
→ They can be chained together to form a pipeline of operations	→ They can't be chained together.
→ pipeline of operations may contain any no. of Intermediate Operations.	→ pipeline of operations can have maximum one terminal operation that too at the end.
→ Intermediate operations are lazily loaded.	→ Terminal operations are eagerly loaded.
	→ They produce end result.

27) What is Peek ?

→ Stream peek() method is an intermediate operation.
→ It takes a Consumer object as an input.
→ It returns a Stream consisting of the elements of current stream.
→ It additionally perform the provided action on each element as elements

Use of Peek :-

→ Peek() exists mainly to support debugging, where we want to see the elements as they flow past a certain point in a pipeline.
→ It is similar to Map, but it takes Consumer object and perform some action on object and return nothing, But map takes a function argument hence apply operation on each element and return the stream having modified elements.

Eg:- SOP(intlist.stream().filter(a→a%2==0).peek(System.out::println).
map(a→a+a).filter(a→a>5).count());

28) What is Reduce ?

→ The Stream.reduce() combine elements of a Stream and produces a single value.
→ reduce operation applies a binary operator to each element in the stream where the first argument to the operator is the return value of the previous application and second argument is the current stream element.

Eg:- SOP(intlist.stream().reduce((a,b)→a+b).get());

29) What are Predicates ?

→ Predicate is a predefined functional Interface (Having Only 1 abstract method).
→ The only abstract method of Predicate is test(T t):
* public boolean test(T t);
→ Whenever we want to check some boolean condition then you can go for predicates.

Eg:- Predicate<String> checkLength = s→s.length()>=5;
SOP("The length of string is greater than 5: "+checkLength.test("mounika"));

30) How to use Predicates ?

→ Say if you need to test if the length of the given string is greater

32) Type parameter and return types of Predicates?
→ Input to predicate can be anything like
Predicate<String>
Predicate<Integer>
Predicate<Employee>
→ Hence only 1 type argument is required which is input type in predicate.
→ Return type is not required as its always Boolean only.

33) Advantages of Predicates?
→ Code Reusability
→ If you have same conditions being used 100 times in a program then you can write once and just use 100 times with checkLength.test(different String to be tested).
→ Conditional checks are holded by functional Interfaces.

35) What is Predicate joining?
→ you can combine predicates in serial predicate.
→ Three ways to join:
AND, OR, Negate
→ Eg: if you want to test 2 conditions:
To check length of string > 5
To check if length is even.

Eg:- Predicate<String> checkLength = s -> s.length() >= 5;
SOP("length of string is greater than 5:" + checkLength.test("mounika")); → true

Predicate<String> checkEvenLength = s -> s.length()%2==0;
SOP("length of string is even" + checkEvenLength.test("mounika")); → false

//it can be joined with and
SOP(checkLength.and(checkEvenLength).test("mounika")); → true && false = false

//it can be joined with or
SOP(checkLength.or(checkEvenLength).test("mounika")); → true || false = true.

//it can be checked with negate.
SOP(checkLength.negate().test("mounika")); → false

34) What are functions?
→ function is also a predefined functional Interface (Having only 1 abstract method).

→ Given some input perform some operation on input and then produce/return result (not necessary a boolean value).

→ This takes 1 input and returns one output.

→ In predicate we used to take 1 i/p and return type is always boolean.

→ In function return type is not fixed hence we declare both i/p type and return type.

Eg:- Function <Integer, Integer> SquareMe = i -> i*i;
SOP("Square of 5 is" + SquareMe.apply(5));

---

31) What is functional chaining?

→ We can combine / chain multiple functions together with "andThen".

→ There are 2 ways to combine functions:

f1.andThen(f2).apply(Input); → first f1 then f2
f1.compose(f2).apply(Input); → first f2 then f1

→ Multiple functions can be chained together like:

f1.andThen(f2).andThen(f3).andThen(f4).apply(Inputs);

Eg:- Function <Integer, Integer> doubleIt = i -> 2*i;
SOP("Double function" + doubleIt.apply(2)); → 4

Function <Integer, Integer> cubeIt = i -> i*i*i;
SOP("Cube function" + cubeIt.apply(2)); → 8

SOP("first Doubling then Cubing" + doubleIt.andThen(cubeIt).apply(2)); → 64

SOP("first Cubing then Doubling" + doubleIt.compose(cubeIt).apply(2)); → 16

---

36) What is Consumer Functional Interface?

→ Predicate <T> takes 1 i/p and returns boolean

→ Function <T, R> takes 1 i/p and 1 return type produced after performing some operations on that input.

→ Consumer <T> it will consume item. Consumers never return anything (never supply), they just consume.

Eg:- take any object and save it's details in Database and don't return anything

```
Interface Consumer<T>{
    public void accept(T t)
}
```

57) What is Consumer chaining?

→ We can combine/chain multiple Consumers together with "andThen".

→ There is only one ways to combine Consumers:

* c1.andThen(c2).apply(input); ⇒ first c1 then c2
* No compose() in Consumer
* Multiple Consumers can be chained together like:
  * c1.andThen(c2).andThen(c3).andThen(c4).apply(Inputs);

```
Consumer<Integer> SquareMe = i -> SOP(i*i);
SquareMe.accept(5); ⇒ 25

Consumer<Integer> doubleMe = i -> SOP(2*i);
doubleMe.accept(5); ⇒ 10

SquareMe.andThen(doubleMe).accept(5);
```

---

58) What is Supplier - functional Interface?

→ Supplier<R> it will just supply required objects and will not take any i/p.

→ It's always going to supply never Consume/take any input.

Eg: always supply me current date

```
Interface Supplier<R> {
    public R get();
}
```

→ No chaining as no i/p is given to this. Only it gives a o/p.

Eg:
```
Supplier<Date> currentDate = () -> new Date();
SOP(currentDate.get());
```

---

59) Use of BiConsumer, BiFunction, BiPredicate and why no BiSupplier?

→ Till now we had:

* Predicate<T> → test() → return boolean
* Function<T,R> → apply() → returns anything
* Consumer<T> → accept() → returns nothing
* Supplier<R> → get() → returns anything

→ What if we need 2 arguments for operation?

→ Then we need BiXYZ - functional Interfaces.

→ There is no i/p in Supplier so no 1 or 2 i/p arguments needed.

Hence no BiSupplier.

```
BiPredicate<Integer, Integer> checkSumOfTwo = (a,b) -> a+b >=5;
SOP("sum of 2 and 5 is greater than 5" + checkSumOfTwo.test(2,5));
SOP("sum of 2 and 1 is greater than 5" + checkSumOfTwo.test(2,1));

BiFunction<Integer, Integer, Integer> multiplyBoth = (a,b) -> a*b;
SOP("multiplication of 5 and 10 is" + multiplyBoth.apply(5,10));
```

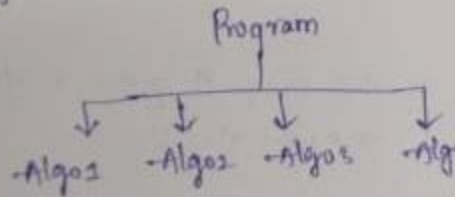
40) If we want to operate on 3 arguments then triPredicate?

→ There are no TriPredicate or TriFunction etc.
→ No QuadPredicate No QuadFunction.
→ Java 8 has inbuilt functional interfaces that can take only 1 or 2 arguments no more.

Introduction to Asymptotic Notations

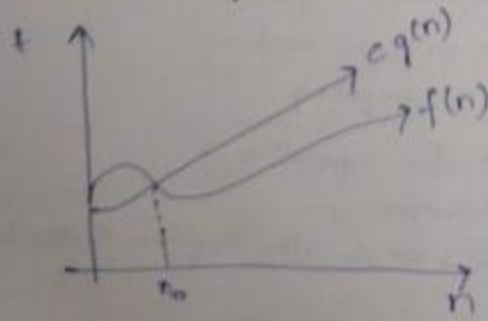
Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For every program we will be having different Algorithms. From that we have to choose an algorithm which has best time complexity and best space complexity.



Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.



$f(n) \leq cg(n)$

$n \geq n_0$

$c > 0, n_0 \geq 1$

$f(n) = O(g(n))$