**JAVA 8**

**Lambda Expression**:

To make instances of anonymous class easier to write and read.  
**What is Java 8 LAMBDA Expressions**:  
 Another way of writing instances of anonymous classes.  
**What is the type of Lambda Expressions**:  
 It is of type functional Interface.  
**Functional Interface**:  
 Functional Interface are nothing but functions with only one abstract method.  
 **Example**: Runnable, FileFilter, Comparator.  
   
 1) Functional Interface doesn’t count in object methods example equals(o) because It is mostly for documentation purpose. It doesn’t override the current method but it redefines it for further implementation.

2) We can even add annotations to functional interface and if the annotations are not functional then the compiler throws an error else everything works smooth.

Can we use Lambda Expression as a variable: Yes, We can use Lambda Expression as a variable and it can be passed as a parameter to other functions and also can be returned back to some function as an variable.  
  
**Is Lambda Expression an object**:  
 No, Lambda Expression is anonymous. It is used to make instance of anonymous class easier. In the old implementation we create an instance with new keyword which in turn allocates some memory but in Lambda Expression no memory is allocated but it is considered as an object in JVM.

New Library got introduces i.e. java.lang.functions.  
We can pass on the data type in the class itself and avoid calling it in the parameters of the Lambda Expressions.

**Method Reference**:  
 In Method reference we specify the Lambda Expression as(:: ). This is nothing but we can specify the class and then lambda expression can be called using this Lambda Expression.  
**Example 1**:  
 **Lambda Expression**:  
 Consumer<String> c = s -> System.out.println( c);

**Using Method Reference**:  
 Consumer<String> c = System.out::println;

**Example 2:**  
 Lambda Expression:  
 Comparator<Integer> c = (i1, i2) -> Integer.compare(i1, i2);

Using Method Reference:  
 Comparator<Integer> c = Integer::compare;

Both Static and Non-Static methods can be used using this method reference concept.  
  
**How can we process our data in Java**:  
 Where are our objects in Java. Most of the time the objects are in a collection(Say List, Set or a Map)  
  
**Using Lambda**:  
 List<Customer> list = //some random list  
 list.forEach(customer -> System.out.println(customer));

**Using Method Reference**:  
 List<Customer> list = //some random list  
 list.forEach(System.out::println);  
Performance of both the implementation is the same but this implementation increase the readability of the code.  
So, we can process the data with Lambda.

We can’t add methods to iterate. This is possible if we add a method to iterate in all the implementation of the interface. So, it is not an option.  
  
The solution for the above problem is use **default methods**. The Advantage of using default methods is that we can write the implementation in the interface itself without breaking the existing implementation. It also allows new patterns of code.  
  
 Interface Iterable<T> {

default public void forEach(Consumer<e> consumer)  
 for(E e: this) consumer.accept(e);  
 }  
Static methods are allowed in JAVA8 interfaces.  
**New patterns in Java8**:  
 Predicates<String> id = s -> s.length() > 20  
 Predicates<String> id1 = s -> s.length() < 10  
 Predicates<String> id3 = id.and(id1)//compare.

Here and is a default method.  
It check whether the predicate is null or not.

**Streams API:**  
  
Suppose we need to compute the average of list of persons whose age is greater than 20.

1. **Mapping**: It takes a list and returns a list.  
   The size of both the lists should be same.
2. **Filtering**: It takes a list and return elements which are filtered.  
   It takes a list and return a filtered list.
3. Find the average of the filtered list.

**What is a Stream:**  
 Stream is a Java Type interface? We can have streams of persons or string etc.…

**What does a Stream do?**  
 It provides an efficient way to process a data. (Especially large amount of data).

**Why it is so efficient?**  
 1) It process the data parallelly. This will be helpful to reduce the computing power of multicore CPU’s.  
 2) Pipelined to avoid unnecessary intermediate computation. (All operations are performed in one pass).

**Why can’t a collection be a Stream?**

This is because Collection API implementation should not be disturbed.  
**What is a Stream?**

1. An Object on which one can define operations. An object that can hold any data.
2. An Object that should not change any data it is processing.
3. It should perform an operation in one pass. It should also process the data in one pass.

**Difference between Intermediary and final?**

1. Any method which return something in Intermediary.
2. Any method which doesn’t return anything is final.

**Reduction:**

The main of Reduce is to perform mathematical operation on streams.  
 There are two kinds of reductions:

1. Aggregation: Min, Max, sum etc... (Classical SQL Operation).
2. Identity of an element is returned when the stream is empty. For Example, if we have empty stream and if we call reduce on it then we will get the result as the Identity element itself.
3. If we have only one element then Identity plus the only element is being returned.
4. These two above scenarios are H cases as Bifunction for reduction has two parameters.
5. What if we are finding max of a stream which is empty. In this case we get **null** as the stream is empty but if we return null, we get NullPointerException. To Handle this there is a new concept called Optional, it is a new class in java to handle the situation like this. We have wrap concept in java which means values in various type like Integer, Float, Boolean etc. Optional in this case represents no value.

**How to use Optional:**

We can make use of optional by checking whether the value is null or not. We can do this in few ways.

1. We have a method called Opt (It is the object of Optional class). Here we can make use of method.isPresent() if it is false then we do nothing else we can return that element by using the method opt.get()
2. We can make use of a default method in Optional Class by making use of method .orElseThrow. This method will return the value if it is present else we can manipulate to throw an exception or else return nothing.

**What are the available reductions?**

1. max(), min(), count()
2. **Boolean Reductions:** allMatch(), noneMatch(), anyMatch()
3. **Reduction that return an optional:** findFirst(), findAny()

Reduction are terminal operations. They trigger the processing of the data. They don’t return any new stream.

**Collectors:**

This is another type of reduction. This is called as <<mutable>> reduction. Instead of aggregating elements, this reduction put them in a container.

After the processing of stream data we can put it in a container using collectors.

**Java Date API (java.time)**

1. **Instant:**
2. Instant 0 is 1st JAN, 1970 midnight GMT.
3. Instant.MIN is 1 billion years ago
4. Instant.MAX is DEC 31 of the year 1,000,000,000
5. Instant.now() is current instant

**Use:**

Instant start = Instant.now();  
 //some computation  
 Instant end = Instant.now();

1. **Duration:**

Duration elapse = Duration.between(start, end);  
long millis = elapsed.toMillis();

A Duration is amount of time between two instants.

1. **LocalDate** to handle random dates.
2. **Period**: Amount of time between two local Dates.

Bridges between legacy API and java.time:

1. Instant and Date
   1. Date date = Date.from(Instant)
   2. Instant instant = date.toInstant();
2. Instant and TimeStamp
   1. TimeStamp time = TimeStamp.from(instant)
   2. Instant instant = time.toInstant();
3. LocalDate & Date
   1. Date date = Date.from(Instant);
   2. LocalDate localdate = date.toLocalDate();
4. LocalTime & Time
   1. Time time = Time.from(instant);
   2. LocalTime local = time.toLocalTime();

TemporalAjusters is useful to get next days week or a month.

**String IO**

* We can create a stream on a string of characters.
* Concatenating a string is costly operation. Thus we use String Buffer for this purpose. In this we can chain the characters to form a StringBuffer. And atlast using toString() to convert it into a String.   
   a) **StringBuffer** is better but it is Synchronized. StringBuilder is another alternative to concatenate String and is faster when compared to String and StringBuffer. StringBuilder is not Synchronized.  
   b) In Java7 concatenation with ‘+’ is by default handled by StringBuilder. So we can avoid using StringBuilder.
* In Java 8 we call **String Joiner**. By using this we create a **String Joiner** using a delimiter and the add the String by using. add method and then convert it to a string then we have easily concatenated the strings. We can also add prefix and postfix.
* Instead of **String Joiner** we can directly call Join method on String class and join the string. We can also pass on String array in this join method with a delimiter.

**String I/O:** We can read text files easily in using I/O of Java. Using the latest **Path** class, we can pass on the directory and then create a stream of this file and again a stream to find something.

* Streams are Auto closable so we do not need to close it explicitly. This will automatically close the file.

**Reading Directories**: Using the same Path class we pass on the directory and using File.list method on the path we walk through each file in the path and check whether it is a directory or not. If it is a directory then we process based on our requirement.

1. Using the above File.list method we can’t go inside the directories. We can do this by using the method File.walk. We can limit the depth by passing on a parameter depth(int) along with the path.

We have got new method for Iteration of list. (Like spliterator, removeIf, replaceAll, sort(by passing a comparator)).

**Comparators:**

.comapring method of Comparator is useful to compare. We can use .theComparing() method to compare over a fetched result. .reverse() for reverse order sorting and .naturalOrder() in normal order.

Hashcode of an Integer is Integer itself.

**New Map Methods:**

**.forEach() –** It takes BiConsumer as a parameter which is nothing but key and value pair.  
 **.get() –** This method fetch the value for a given key. But what is the reason behind it returning a null. Whether it is null or has a value null. To handle this, we have a new update we pass a default value and call getorDefault() so if value is present then we get a value or else a default value is returned.

.put(), replace() and remove(), compute(), computeIfPresent(), computeIfAbsent()

**Annotations:**

Java 8 has new concept to add multiple annotations. Suppose there is a test case where in we need to pass multiple annotations then Java8 has enable this feature. This can be done by wrapping annotations in Java 7 but in Java 8 we can write multiple annotations. We can add Repeatable annotation to tell that this annotation is repeatable.

Java8 also allows annotation to be kept on types.

**Java FX 8:**

This is helpful for interface, controllers and handling callbacks.

**A Simple Example:** We create a class which extend Application and override start() method. We use launch() method on the class to start the application.

**Stage**: A top level of a window  
**Scene**: A Scene must hold a stage, and a stage must reside in a stage. A Scene holds all the graphical components, shapes etc.

In our stage we add a label to the scene and then scene is added to stage and then show method will show the top level window.

**REPL: Read, Eval, Print and Loop**jjs is the REPL executable it is located in javac

//print 2d array using streams

Arrays.stream(trips)

.flatMapToInt(a -> Arrays.stream(a))

.forEach(System.out::println);