Java 8

Link :- https://www.youtube.com/playlist?list=PLd3UqWTnYXOlrKZWFTbgguqNRA_uVyeBl
Udemy :- https://idemia.udemy.com/course/modern-java-learn-java-8-features-by-coding-it/

Java 8 Release Date :- 18th-Mar-2014

Why Java 8 :- Concise the code by enabling functional programming and to enable parallel programming.

Java 8 new features :-

1. Enabled functional programming via lambda expressions.
2. Lambda Expression.
3. Functional Interfaces.
4. Default Methods and Static Methods.
5. Predefined Functional Interfaces.
6. Double Colon Operator (::)
7. Streams.
8. Date and Time APIs OR JODA APIs
9. Optional Class.
10. Nashorn Java Script Engine. etc.

Lambda Expression :-

- LISP is the first programming language who used lambda expression
- Lambda Expression are not related to anonymous inner classes.
- No separate .class file will be generated for lambda expression.
- Anonymous classes are more powerful then the Lambda Expression because Lambda Expression can not be used without Functional Interface.

Main objective of lambda expression :- To bring benefits of functional programming in java.

What is the lambda expression :- It is an anonymous (Name Less OR No Name) function. Without name, without any return type and without modifiers.

Ex 1 :-

Without Lambda :-

```
public void printHello(){
        System.out.println("Hello Lambda");
 }
```

Converting it in lambda :- (Just remove name , return type and access modifier, and put an arrow)

```
() -> { System.out.println("Hello Lambda"); }
```

Ex 2 :-

Without Lambda :-

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

Converting it in lambda :- We do not need to specify the type explicitly. Compiler will able to understand the type of the arguments.

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

Ex 3 :-

Without Lambda :-

```
public int squreIt(int n){
   return n * n;
}
```

Converting it in lambda :- If there is a single line of code in curly braces then we do not need to use return statement. It is optional.

```
(n) -> { return n * n; }
(n) -> { n * n; }
```

If there is single argument we are passing to function then parenthesis are optional. So final lambda expression is as follows

```
n ->  n * n;
```

Note :-

- Without curly braces we cannot use return statement. If we use it then its an compilation error. Compiler will automatically consider a return statement.
- Within curly braces if we want to return some value then we have to use return statement.
- Examples :-
  - n -> return n * n; -> Invalid (Return statement should be in parentheses)
  - n -> { return n * n; }; -> Valid
  - n -> { return n * n; } -> Valid
  - n -> { return n * n }; -> Invalid (Semicolon is missing)
  - n -> n * n; -> Valid
  - n -> { n * n }; -> Invalid
  - n -> { n * n; }; -> Invalid (Return statement is missing)

Functional Interface:-

Definition :- A interface with single abstract method is functional interface. This interface can contain static and default method.

We should use **@FunctionalInterface** annotation to specify explicitly that a particular interface is functional interface.

Rules of Functional Interface :-

- Functional interface can hold only one abstract method.
- It can hold N number of static and default methods.
- Functional Interface WRT inheritance :-

Ex 1 :-

```
@FunctionalInterface
interface A{
    public void a();
}

@FunctionalInterface
interface B extends A{

}
```

Ex 2 :-

```
@FunctionalInterface
interface A{
    public void a();
}

@FunctionalInterface
interface B extends A{
     public void a();
}
```

Ex 3 :-

```
@FunctionalInterface
interface A{
    public void a();
}

@FunctionalInterface
interface B extends A{
     public void b();
}
```

- If parent interface is Functional Interface then child interface is also an functional interface if there is no abstract method in child. Like Ex 1.
- If parent class abstract method and child class abstract method has same name then it is functional interface. Like Ex 2.
- If both parent and child has different abstract method then these are not functional interface.

- Functional Interface can be used to provide reference to lambda expression to execute it. For ex :- HelloLambda hl = () -> {System.***out***.println("Hello Lambda");};

Note :-<u>Without Functional Interface we can not write Lambda Expression.</u>

Default Method :-

1. Also known as virtual extension method and defender method.
2. Without effecting the implementation of classes who are already implemented a interface, if we want to add a new function in interface then we have to make it default function otherwise it is an compilation error.
3. Object class can be declared in interfaces as default method. We are not allowed to override them in interface.
4.

Predefined Functional Interfaces :-

1. <u>Predicate</u> :-
    1. To do the conditional checks we can use lambda expressions with Predicate.
    2. There is a function in Predicate interface, who is responsible to perform this is as follows :-
        - public abstract Boolean test(T t);
    3. Predicate is useful when we want put diff-2 checks on diff-2 conditions then we will create diff-2 predicates and then we can use wherever we want to use and any number of time. So it will save line of code.
    4. <u>*Predicate Joining*</u> :
        1. We can join two or more predicates together to perform multiple check.
        2. Java support and() , or() , negate() functions in predicate.
2. <u>Function</u> :-
    1. Predicate always return boolean as result but if we want some specific result then we can go for Function interface.
    2. There is a function in this interface, who is responsible to perform all such tasks is as follows:-

      - public abstract R apply(T t);
   3. Function chaining is also applicable. For ex :-
      - f1.andThen(f2).apply(i)   --> f1 will apply first and then over the result f2 will apply.
      - f1.compose(f2).apply(i)   --> f2 will apply first and then over the result f1 will apply.
3. Consumer :-
   1. Consumer will always accept an input and then only do any operation and never going to return anything.
   2. There is a function who is responsible for it is:-
      - public abstract void accept(T t);
   3. We can also create chain of Consumer via andThen() method.
4. Supplier :-
   1. If we just want to supply an object then we use Supplier. It do not take any input for it.
   2. It contains only one method that is responsible to supply the object. That is :-
      - public abstract T get();
5. BiPredicate :-
   1. BiPredicate is used to get 2 arguments and perform some operation on that.
   2. Rest of the functionality is similar to Predicate.
   3. The function is:-
      - public abstract boolean test(T t, U u);
6. BiFunction :-
   1. BiFunction is used to get 2 arguments and perform some operation on that.
   2. Rest of the functionality is similar to Function.
   3. The function is :-
      - public abstract R apply(T t, U u);
      - BiFunction<Integer, String, Employe> biFun = (salary, name) -> new Employe(salary, name);
7. BiConsumer :-
   1. BiConsumer is also used to get 2 arguments and perform some operation and then show or store the result.
   2. The function is :-
      - public abstract void accept(T t, U u);
      - BiConsumer<Employe, double> biCon = (e, sal) -> {e.sal = e.sal + sal; Syso(e);};

8. <u>IntPredicate</u> :-
    1. If we need to perform predicate operations over int values then we should always go for IntPredicate.
    2. It enhance the performance by saving process time. (Boxing and Unboxing)

Method and Constructor reference OR Double Colon Operator (::) :-

1. <u>Method Referrence</u> :-
    - It is an alternative syntax for lambda expression.
    - Code re-usability is enhanced.
    - If implementation is already exists then we can go for method reference.
    - Argument list should be same as calling function.
    - If the return type of functional interface's function is void then after implementation we can convert it in any return type. For ex :-
        - public void run();   -> Runnable Interface
        - private int m1();   -> Run method can be converted like m1() method.
    - But Vice versa is not possible. For ex:-
        - public int sum(int x, int y)   -> Functional interface method.
        - private void doTheSum(int x, int y);   -> Here return type should be same as sum() method.
    - There are multiple syntax of it. For ex.
        - If we are calling a static method of class :-
            - ClassName :: methodName; --> String s = HelloJava :: hello;
        - In case of instance methods :-
            - objectReference :: methodName --> String s = new HelloJava :: hello;
        -
2. <u>Constructor Reference</u> :-
    - If method implementation is not available then we go for constructor reference.
    - Syntax :-   <Name_Of_Class> :: new ----> ex :- Test :: new;
    - It create new object of a class each time.

Streams :- (https://www.youtube.com/watch?v=5duxFiseLRE)

1. If we want to process object from collection then we go for streams.
2. Stream is not a data structure like ArrayList etc.

3. Stream do not change original data source's data. It simply process this data and perform operation over it.
4. As a result, always a new object is received.
5. We can not iterate stream more then once. But Collection can be iterated multiple times with the help of for loop and Iterator interface.
6. Following are the intermediate operations on streams :-
    - Filter -> It always take argument of Predicate<T> functional interface.
    - Map -> It always take the argument of Function<T,R> functional interface.
    - sorted -> It take argument of Comparator functional interface.
    - flatMap ->
7. Following are the terminal operations on Streams :-
    - Collect -> Collect is used to collect the processed data of the stream.
        - joining:- This will join all the inputs.
            - joining(): Concate the stream.
            - joining(delimiter): Concate the stream by inserting delimiter among the stream elements.
            - joining(delimiter, prefix,sufix): Concate the stream by inserting delimiter among the stream elements and adding prefix,sufix to generated value.
        - counting :- This will return the count of resutl that a stream can return.
        - mapping :- This is similar to map()
        - groupingBy :- This will group the data with respect to a key. At the end a map will be returned.
            - groupingBy(classifier):- Group data with respect to given classifier. A Map<classifierType, List<StreamType>> will be returned.
            - groupingBy(classifier, downstream):- downstream means any other collector like Collectors.toSet(), Collectors.groupingBy(), Collectors.summingInt() etc.
            - grouingBy(classifier, supplier, downstream): supplier determines the type of result that is going to be returned.
        - partitioningBy :- It is also a kind of groupingBy(). But it groups the data as per the supplied predicate.
            - partitionBy(predicate):- It returns a Map<Boolean, List<StreamType>>. The values that is satisfied by predicate will come under the true key of map otherwise it will be under the false key of map.

      - partitionBy(predicate, downstream): downstream means any other collector like Collectors.toSet(), Collectors.groupingBy(), Collectors.summingInt(), Collectors.partitionBy() etc.
    - reduce :- Used to reduce the content of a stream in a single value.
  - forEach -> It use to iterate the result of stream.
  - reduce -> It is used to reduce the result in single value.
8. Other Operations :-
   - min()
   - max()
   - toArray()
9. We can also create streams of non-collection concepts too. If we have group of object like array then we can convert it in stream via **Stream.of()**;

Optional :-

1. Optional is used to handle NullPointerException.
2. Optional check whether the value is there in stream or not.
3. Creating Optional :-
   - Optional.empty() :- create empty optional.
   - Optional.of(T t) :- Create Optional of given type.
   - Optional.ofNullable(T t) :- create optional of a value and take care whether the value is null or not.
4. We can apply both filter() and map() methods on Optional as well as streams.
5. isPresent() method is used to check whether a value is exists or not.
6. ifPresent() is alternate of isPresent(). It takes an Consumer<T> argument.

Nashron :-

1. Nashron is used to call the js from command line and from a java program file and vice versa.
2. java 8 provided **jjs** command to run js file from command line.
   - jjs <jsFileName>
   - jjs helloJavaJs.js