Java 8 Features

- Lambda expression
- Functional Interfaces
  - Use to call lambda expression
- Default methods in interface
- Static methods in interface
- Functional Interface
  - Predicate
  - Function
  - Consumer
- :: operator for method reference and constructor reference

- Stream API
  - Useful for operation on collection
- Date and Time API
  - Called as Joda API
  - Added in joda.org

Why java 1.8

- To simplify programming
- To get advantages of Parallel processing
- To get benefits of functional programming lambda functions got added

Why Lambda expression

- To enable functional programming in JAVA
- Reduces size of code, more readable and maintainable
- To use API very effectively
- We can assign a function to a variable
- Functions can be passed as parameter to other functions and constructor

- Lambda expression can access class variable in which it is written
- Lambda expression can access local variable of functions in which it is written
- But the method local variable referenced from lambda expressions implicitly treated as final. So you cannot modify its value. It’s a compile time error

Default methods in Interface

- Interfaces allow to add methods with implementation
- Interfaces allows to add static methods also.
- Functional Interfaces are added
- If a class implements 2 interfaces, both contain a default method with same name then it will give compile time error
- To resolve that add our overridden method.
- But if you want default method of MyInterface1 that can be used
  - MyInterface1.super.mymethod();

```
public interface MyInterface1 {
 default void m1() {
 System.out.println("This is from MyInterface 1 method 1")
 }
}

public interface MyInterface2 {
default void m1() {
 System.out.println("This is from MyInterface 2 method 1");

 }
}
```

- Overrode m1 otherwise its compile time error.

public class MyClass implements MyInterface1,MyInterface2 {

//will show compile time errors because of both Interfaces have same default method

//if you override method will work

public void m1() {

System.***out.println("My overriden method");***

MyInterface1.**super.m1();**

//Method from particular interface can be wrapped in side overridden method

}

}

Differentiate abstract class and Interfaces

Interface with default Method	Abstract Classes
Every variable is public static final	Variables can be static or instance variable
Interface doesnot have state of object because it doesnot have any instance variable	But abstract call can have state. Instance variables are allowed
Functional Interface with default methods can refer to lambda expressions	Cannot refer lambda expression
We cannot override Object class methods	We can override Object class methods
Cannot declare instance and static class as nested class	Can declare instance and static class as nested class
Constructors cannot be written	Constructors can be wriiten

Predefined Functional Interfaces

- Consumer
- Supplier
- Function
- Predicate
- All are defined in package java.util.function

Predicate Interface

- Predicate can take some input parameter perform some operation and written some Boolean value.
- Predicate must accept some value as parameter

```
 interface Predicate<T>
{
  boolean test(T t);
}
```

Function Interface

- Function takes some input parameter. Perform some operation. Written some o/p but return type need not be Boolean

```
Interface Function<T,R>{
 R apply( T t)
}
```

- Consumer accepts some values but does not return anything
- It only consume the value
- Interface Consumer<T>

```
{
  void accept(T t)
}
Consumer<String> c=s->System.out.println(s);
c.accept(“Hello”);
c.accept(“World”);
```

Supplier interface

- Supplier interface will always provide the value
- So it will always return something but does not accept any parameter
- It is opposite of consumer

```
Interface Supplier<R>{
  R get()
}
Supplier<String> s=()->{String[] str={“Hello”,”Welcome”,”greet”};
                                int x=(int) Math.random()*2+1;
                                return str[x];
                               }
System.out.println(s.get());
System.out.println(s.get());
```

Stream functions

- Stream functions are used to process objects from collection
- To create stream from a collection object we may use stream function
- Stream function is added in Collection Interface in java 1.8
- So can be used on any collection
- Stream is a interface
- Stream s=c.stream();

```
ArrayList<Integer> lst=new ArrayList<>();
lst.add(0);
lst.add(20);lst.add(10);lst.add(5);
System.out.println(lst);
```

- To list all even values in another list

Without stream

```
List<Interger> lst1=new ArrayList<>();
for(Integer i:lst)
{
    if(i%2==0)
       lst1.add(i)
}
```

- With Stream

```
List<Integer> l1=lst.stream().filter(i->i%2==0).collect(Collector.tolist());
System.out.println(l1);
```

- filter(Predicate<T> t)
- map(Function<T,R> f)
- collect()
- count()
- sorted(Comparator c)
- min(Comparator c)
- max(Comparator c)
- forEach(lambda expression)
- toArray() – to conver inarray
- Stream.of()

- Initially we were using Date, Calendar, TimeStamp classes , but these are not efficient. Most of them are deprecated
- Date and Time API newly added in java 1.8 to improve performance of handling Date and Time values
- These newly added API are also called as Joda Time API because these are introduced by joda.org
- Example to display current System date and current System time

```
LocalDate dt=LocalDate.now();
System.out.println(dt)
```

- LocalTime lt=LocalTime.now();
- System.out.println(lt);
- These are available in java.time package
- //to retrieve part of date day, month and year
- **int dd=ld.getDayOfMonth();**
- **int mm=ld.getMonthValue();**
- **int yy=ld.getYear();**
- System.***out.printf("%d-%d-%d",dd,mm,yy);***

- LocalTime lt=LocalTime.now();
- Int hr=lt.getHour();
- Int m=lt.getMinute()
- Int s=lt.getSecond();
- Int ns=lt.getNano ()
- System.out.printf(“%d:%d:%d:%d”,hr,m,s,ns);