Java 8 Features

In order to compete with Python,R Java introduced Consise Code(less code to do more things) by enabling functional programing i.e Lambda Expression.

1. Lambda Expression
2. Functional Interface
3. Default methods and static methods
4. Predefined functional Interface-> predicate,Function,Consumer,Supplier
5. Double colon operator(::) -> Method reference, Constructor reference
6. Streams
7. Date and Time API
8. Optional class
9. Nashorn Javascript Engine

1. Lambda Expression

Lambda Expression is already present in other languages.
The Main objective of Lambda Expression is to bring the benefits of functional programing in Java.

What is Lambda Expression:
It is an anonymous (name less) function, without return type, without modifiers.
It is represented by arrow symbol **->**

Eg: How to write lambda expression

Original method	lambda Expression

Public void m1(){
    Sop("hello");
}
	()->{Sop("hello");}
	If the body of Lambda Expression Contains only one line then, curly braces are not needed.
	It is Mandatory for multiple lines.
	()->sop("hello");

Public void m1(int a,int b){
    Sop(a+b);
}
	If the compiler can guess the type of parameter then we need not specify the parameter type explicitly
	(a,b)->sop(a+b);

Public int squareIt(int n){
  return n*n;
}
	(int n)-> {return n*n;}
	(int n) -> n*n;
	The return word is mandatory if curly braces are used.
	Also if compiler can identify the type then
	(n)->n*n;
	Also if there is only one parameter then parenthesis is not needed
	n->n*n

Public void m1(String s){
   return s.length();
}
	s->s.length();

Inorder to call the lambda expression we make use of Functional Interface(FI)

Summary:

- A lambda expression can have any number of arguments.
- If there are multiple arguments these should be separated by comma eg: (a,b).
- If there is only one argument parenthesis is optional.
- If the compiler can guess the types automatically, type is not necessary.
- If the body contains only one line then curly braces are optional unless using return keyword.
- Without curly braces we cannot use return. Compiler will consider return value automatically.
- Within curly braces if we want to return some value, compulsory we should use return statement.

```
n->return n*n;   =➔Invalid
n->{return n*n;}; =➔Valid
n->{return n*n}; =➔Invalid
n->{n*n} =➔ Invalid
n->n*n; =➔ Valid
```

2. **<u>Functional Interface</u>**

An interface is said to be a functional interface if it contains only one (single) abstract method.
Eg: Runnable(run()), Comparable(compareTo()), Comparator(compare()), ActionListener(actionPerformed()), Callable(call()).
In order to invoke lambda expression Functional Interfaces are necessary.

There can be any number of static and default methods present in interface, the restriction is applicable only for abstract methods.

We use annotation @FunctionalInterface.

Eg: @FunctionalInterface
```
Interface A{
        Public void m1();
}
```

@FunctionalInterface
Interface A{
    Public void m1();
}
 	
@FunctionalInterface
Interface B extends A{
}
	This is valid since method m1() is by default available in interface B.

@FunctionalInterface
Interface A{
    Public void m1();
}
 	
@FunctionalInterface
Interface B extends A{
    Public void m1();
}
	This is Valid, Since m1() is overridden in interface B and ultimately it contains only one abstract method.

@FunctionalInterface
Interface A{
    Public void m1();
}
 	
@FunctionalInterface
Interface B extends A{
    Public void m2();
}
	This is Invalid, since Interface B contains more than one abstract method.

@FunctionalInterface
Interface A{
    Public void m1();
}
 	
Interface B extends A{
    Public void m2();
}
	This is valid, since child is not a functional interface.

Conclusion: If the parent class is Functional interface, then child class is also functional interface if and only if it does not define any new abstract method.

Example:

```
Interface Interf{
        Public void m1();
}
Class Test {
        Public static void main(String args[]){
                Interf i=()->SOP("hello");
                i.m1();
        }
}
```

i.m1() is the call to lambda expression.
To provide reference to lambda expression functional interface is required.

When we use functional interface compiler can automatically identify the type of arg ument based on abstract method in our functional interface.

Eg:

```
Interface Interf{
        Public void add(int a, int b);
}
Class Test{
        Psvm(string args[]){
                Interf i=(a,b)->sop("the sum is"+(a+b));
                i.add(10,20);
                i.add(100,200);
        }
}
```

There will not be any separate .class file generated for lambda expression

<u>Lambda Expression in Multithreading:</u>
We can implement a thread by using Runnable interface, Runnable interface is functional interface, hence we can implement multithreading using lambda expression.

```
Class Test{
        Public static void main(String args[]){
                Runnable r = ()->{
                        For(int i=0;i<10;i++){          Lambda expression
                                Sop("child Thread);
                        }
                };
                For(int i=0;i<10;i++){
                        SOP("main Thread");
                }
        }
}
```

<u>Lambda Expression in Collections:</u>
In collections there is an interface comparator, which we can use for sorting. This interface is a functional interface since it has only one method compare(), thus we can use lambda expression.
About compare(): it returns a int value based on below
Int compare(Object obj1, Object obj2)

- -ve if obj1 has to come before obj2
- +ve if obj1 has to come after obj2
- 0 if obj1 and obj2 are equal

```
Class Test{
        Psvm(String args[]){
        ArrayList<Integer> l =new ArrayList<Integer>();
        l.add(10);
        l.add(5);
        l.add(20);
        l.add(0);
        Comparator<Integer> c=(I1,I2)->(I1<I2)?-1:(I1>I2)?1:0;
        Collections.sort(l,c);
        SOP(l);
        }
}
```

Anonymous Inner class vs Lambda Expression:
Anonymous Inner class can work with Interfaces having multiple abstract methods.
Anonymous inner class is more powerful.

3.Default and static Methods:

Every method present inside interface is public and abstract until ver 1.7
From ver1.8 default and static methods are allowed.

Every variable is public static final (no enhancements made)

Default methods is also known as virtual extension method or defender method.
If an interface which is implemented by multiple classes want to add an method then it becomes mandatory to update all the classes, inorder to avoid this default method concept came into picture.
Syntax: default void m1(){}

If implementation of default method is not satisfied we can override it.
While overriding the modifier should always be public.
Object class methods cannot be implemented as default methods.

We can implement 2 interfaces containing same default method as long we are overriding in our class.

Similar to default we can have static method in interface.
Also we can write main method
Static methods should be called with the interface name

Abstract class vs Interface

Conceptually both are same, but we can implement multiple interfaces.
Abstract class variable can have any modifier.
Interface are public static final.
Intialization of variable is mandatory in interface.
Interface cannot have constructor

<u>4.Predefined Functional Interfaces:</u>

Predicate<T> → boolean method Test()
Function<T,R> → R type method apply()
Consumer<T>→ Void method accept()
Supplier<R> → similar to get R is return type method get()
<u>Conditional checks</u>
Predicate is an functional interface which contains only one method boolean test(), in order to use lambda expression we need the predicate

Eg: Predicate<Integer> p1=i->i%2==0;

Predicates are needed for conditional checks we done need to return true or false explicitly.
To use predicate on multiple elements we can do
P1.test(element);

Also we can use join predicates
p1.and(p2).test(element)
p1.or(p2), p1.negate(p2)

Ideally predicate is like defining a test method

<u>Perform Operation:</u>
Functions
the interface function consists of one method apply
this compulsory needs two arguments, input type and return type

eg: Function<integer,integer> f = i->i*i;
f.apply(2);
f.apply(4); and so on..

similar to above we can combine two functions
f1.andThen(f2).apply(i) --→ f1 follwed by f2
f1.compose(f2).apply(i) --→ f2 followed by f1

<u>Void functions:</u>
Consumer interface is used for this purpose, it has only one method accept()

And works similar to void functions
Example: Consumer<String> c=s->sop(s);
c.accept("ABC");

also consumers can be chained by using c1.andThen(c2).accept();

<u>No input but output</u>
we can use supplier function.

Example: Supplier<Date> s = ()->new Date();

Note:- There are other functions too but not considered here.

<u>Methods and Constructor reference:</u>
Alternative to Lambda expression

Classname::method name for static method
Object::method name for non static
We can directly assign this to functional interface.

Example:
```
Public static   void m1(){
        For(int i=0;i<10;i++){
                System.out.println("child Thread");
        }
}
Psvm(string[] args){
        Runnable r= Test::m1;
```

M1() and Run() argument must be same

When mehod returns a object type we should use constructor reference.

Eg: Interf i=Sample::new;

Interface method
Public Sameple get();

In case of arguments method will call paramterized constructor;