JAVA 8

Java 8 was released in the year 2014 as a major version.

The main aspects of java 8 is

→To simplify the programming

→To enable functional programming.

→Affectively utilizing system resources

Features of java 8: -

1. Default methods in interface
2. Static methods in interface
3. Functional interface
4. Lambda expression
5. Method reference & Constructor reference
6. Stream API
7. Date and Time API

Functional Interface:

The interface which exactly contains only one abstract method is known as Functional Interface.

In Functional interface restriction is applicable for only abstract methods

A functional interface can contain more than one default method or static method.

@FunctionalInterface annotation will help to provide information about functional interface to compailer.

Functional Interfaces are mainly introduced to use lambda expression in java.

Ex:- Runnable run();

Callable call()

Comparable compareTo()

Comparator compare()

Note: - In functional interface we can declare object class methods as abstract.

Lambda Expression: -

Lambda expression is an anonymous function which does not contain modifiers, return type and method name.

Lambda expression is nothing but implementation of abstract method of functional interface.

Lambda expression is suitable only for functional interface implementation.

Functional interface declaration only can hold lambda expression.

```java
interface I1{
    void m1();
}

class I1impl implements I1{

    @Override
    public void m1() {
        System.out.println("hello");
    }

}
public class Driver {

    public static void main(String[] args) {
        I1 obj = new I1impl();
        obj.m1();

        I1 obj2 = ()->{System.out.println("Hello");};
        obj2.m1();//lambda impl
    }
}
```

Parameterized: -

```
public interface Calculator {
    int calc(int a, int b);
}

public class App {

    public static void main(String[] args) {

        Calculator addition = (int a, int b)->{
                              return a+b;
                              };
        System.out.println(addition.calc(10,5));

        Calculator subsraction =  (x,y)->{
            return x- y;
        };
        System.out.println(subsraction.calc(10,5));

        Calculator multiply = (x,y)-> x * y;
        System.out.println(multiply.calc(2, 5));

    }
}
```

Rules to declare Lambda expression: -

- ➔ The number of arguments present in abstract method of functional interface and number of arguments declared in lambda expression must be same.
- ➔ In lambda expression we need not to provide data types for arguments.
- ➔ If lambda expression contains single argument defining parenthesis () is optional.
- ➔ If lambda expression contains single statement then defining curly braces is optional {}.

➔ Lambda expression contains only 1 statement as return statement then defining return keyword is optional but here we don't have to provide curly braces.

```
interface Square{
    int area(int a);
}
public class Driver {

    public static void main(String[] args) {
        Square s = x -> x *x;
        System.out.println(s.area(5));
    }
}
```

Method Reference: -

By using method reference we can provide implementation to abstract method of functional interface.

In place of lambda expression, we can use method reference.

In method reference we map existing method implementation to abstract method of functional interface.

```
interface B{
    int hello();
}
public class TestA {
    public int m3() {
        System.out.println("m3 executed");
        return 45;
    }

    public static void main(String[] args) {
        TestA ref = new TestA();
        B obj5 = ref::m3;

        System.out.println(obj5.hello());
    }

}
```

NOTE:

In method reference concept ther is no restriction on modifiers and return type , only there is restriction on number of arguments of reference method.

- ➔ If abstract method has return type then reference method should have the same return type or child type should be there.
- ➔ If abstract method contains void as return type , then we can use any return type in reference method.
- ➔ If abstract method contains return type either primitive or non-primitive, tehn we have to use same return type for reference method.

Constructor reference:

If abstract method of functional interface contains non-primitive return type, in such scenario we can go with constructor reference.

```
interface C{
    Car getCar();
}
class Car{
    public Car() {
        System.out.println("Constructor executed");

    }
}
public class TestC {
    public static void main(String[] args) {
        C obj = Car::new;
        System.out.println(obj.getCar());
    }
}
```

Stream API: -

Stream API is introduced in java 1.8 version. This will help process the objects of collection or it will help to perform bulk operations on collection object.

Stream:

Stream is an interface present in java.util.stream package.

This interface contains multiple method which helps to performs some operations or collection object.

Methods of Stream interface: -

filter(), map(), collect(), toArray(), sorted(), max() , min() etc.,

stream(): -

➔ stream is a method which is declared in collection interface.
➔ Stream () is a default method which will give stream type object.
➔ It is introduced in 1.8 version.

filter(): -

➔ filter() is a method of Stream interface which helps to filter the elements of stream based on condition declare in lambda expression.

collect(): -

collect() is a method of Stream interface helps to collect the elements of shown into collection based on given arguments.

Collectors: -

Collection is a class present in java.util.stream package .

This class contains some static method which helps to convert stream elements into collection elements.

toList(), toSet(), toMap() etc.,

```
List<Integer> list = Arrays.asList(10, 20, 1, 3, 4, 22, 55);
List<Integer> list3 = list
        .stream()
        .filter(e -> e %2 == 0)
        .collect(Collectors.toList());
System.out.println(list3);
```

map(): -

map() is a method which helps to modify the elements of stream.

Ex: -

```
List<Integer> list = Arrays.asList(10, 20, 1, 3, 4, 22, 55);

List<Integer> list2 = list
                        .stream()
                        .map(a -> a * 2)
                        .collect(Collectors.toList());
System.out.println(list2);
```

count(): -

count() is a method in Stream interfaces which helps to find number elements present in stream

ex: -

long c = list.stream().count();

System.out.println(c);