JAVA-8 Features

JDK Version-8 features *

(1) Functional Interface.
(2) Default keyword
(3) Lambdas Expression.
(4) Stream API → Collection.
(5) Optional class → Exception in Java

(1) **Functional Interface** Functional Interface is an Interface that should consist of actally only one incomplete/Abstract method in it.

This features was introduced into Version 8 of Java

* It is common

Example of Runnable Interface:

Ex-1

```
Package P1;
@FunctionalInterface.
Public Interface A {                    → error
    Public void test();                 → After use error gone (A)
}
```

Ex-2

```
Package P1;
@FunctionalInterface
Public Interface A {
    Public void test1();
    Public void test2();
}
```

→ होनही होना चाहिए
होगा तो Error आएगा।

② Default Keyword :- Default Keyword was introduced in version 8 of Java.

* Using Default Keyword we can develope Complete method in the Interface.

Ex- A.java

```
Package P1;
Public Interface A{
    default public void test1()
    {
        Sop("1");
    }
    default void test2()
    {
        Sop("2");
    }
}
```

B.java

```
Package P1;
Public class B implements A{
    psvm()
    {
        B b1 = new B();
        b1.test1();
        b1.test2();
    }
}
```

// 1, 2

Note:- In the Functional Interface one incomplete method is mandatory to write but any no of Complete method should be present - if we override.

Ex-

A.java

```
package P1;
@FunctionalInterface
public Interface A{
    public void test();
    default public void test1()
    {
        Sopln(100);
    }
}
//
```

B.java

```
package P1;
public class B implements A{
    @Override
    public void test(){
        Sopln();
    }
    psvm(){
        B b1 = new B();
        b1.test();
        b1.test1();
    }
}
```

// 1, 100

③ Lembda Expression ← Lembda's Expression was introduced in version-8 of Java.

* Using Functional Interface we can Reduce the No of Codes - in our programs.

Ex-1

```
Package P1;
@FunctionalInterface
Public Interface A {
    Public void test();
}
```

No implement
No - overridding.
No - Object creation

```
Package P1;
Public class B {
    Psvm()
    {
        A a1 = () -> {
            Sop ln(100);
        };
        a1.test();
    }
}
```

lembda Exp.

// 100

Ex-2

```
Package P1;
@FunctionalInterface
Public Interface A {
    Public void test(int x);
}
```

```
Package P1;
Public class B {
    Psvm()
    {
        A a1 = (int x) -> {
            Sop ln(x);
        };
        a1.test(100);
    }
}
```

// 100

Ex-3

```
Package P1;
@FunctionalInterface
Public Interface A {
  Public void test(int x);
  default void test1() {
    Sop(100);
  }
}
```

```
Package P1;
Public class B {
  psvm() {
    A a1 = (int x) -> {
      Sopln(x);
    };
    a1.test(100);
    a1.test1();
  }
}
// 100, 100
```

④ Stream API * Stream API are related to Collection framework which was introduced into Java 1.8 version

* Java Stream API is very different from the I/O Stream.

* Stream API reduced the code length.

* Stream API is basically perform bulk operations and process of object of Collection.

* Stream API is used to process of Collections of objects.

* A Stream is a sequence of objects that supports various methods which can be pipelined to produced dissered the output.

* filter & map methods with Example
* Other Pupertant methods and operations Such as- Collect(), sorted(), min(), max(), forEach(), toArrays(), Stream.of().

* Package PL;
Public class StreamMain {
PSVm() {
// Create a list and filter all even number from list -

All Common

method-L to make List

List<Integer> list1 = List.of(2,4,50,21,22,67);

make local variable. Unmodefiable

X list1.add(45); — Can't add list.of is
Sop(list1); // error Unmodifiable.
Sop(list1); => => 2,4,50,21,22,67

}

}

method 2 To create List:

VVI `List<Integer> list2 = new ArrayList<>();`

```
list2.add(12);
list2.add(34);
list2.add(23);
list2.add(78);
```

m-3 to create List `List<Integer> list3 = Arrays.asList(23,567,12,677);`

↓

Unmodifiable.

&

Can't add.

* Find out the even Number from ListL

* Old method (without Streams)

```
List<Integer> listEven = new ArrayList<>();
    for(Integer i: listL){
        if(i%2==0){
            listEven.add(i);
        }
    }
    Sop(listL);
    Sop(listEven);
}
}
```

Using Stream API :-

```
Stream<Integer> stream = listL.stream();
                  ↓ local
List<Integer> newlist = stream.filter(i -> i%2==0).collect(Collectors.toList());
   ↓ Local variable      ↓ make it local
   Sop(newlist);
  }
}
```

⇓

The above can also make into a single line.

* **Sort the Number from list1 which is greater than 10.**

```
List<Integer> list2 = list.stream().filter(i -> i > 10).
                      collect(collector.toList());
```

→ make local variable (ALT + Enter)

```
Sopln(list2);
}
}
```

Result = 50, 21, 22, 67

* **How to create Stream Object :-**

```
Public class StreamObject {
    PSVM() {
    // Stream API - Collection process
    // Collection / group of Objects.
```

Default in All

(i) Blank Stream:-

```
Stream<object>s = stream.empty();
        EmptyStream
    emptyStream.forEach(e->{sopln(e);
                            });
        }
    }
```

(2)

```
String nam[] = {"Durgesh", "Uttam", "Ankit", "Divya");

    Stream<String>Stream1 = Stream.of(Names);

        Stream1.forEach(e->{
                    sopln(e);
                });
        }
    }                    // Durgesh, Uttam, Ankit, Divya.
```

3. Builder Method:-

```
Stream<object> StreamBuilder = Stream.builder().build();
```

4. To work on List, Set :

```
List<Integer> list2 = new ArrayList<>();
    list2.add(42);
    list2.add(55);
    list2.add(48);
```

```
list2.add(78);
Stream<Integer> stream2 = list2.stream();
        stream2.forEach(e -> {
                Sopln(e);
            });
    }
}
```

* Important methods which Mainly Used *

```
Package P1;
Public class methods { } -> Common in all.
    psvm() {
```

// Create List:- // filter (Predicate)
// Boolean value function. True/false
(पर कंडीशन होगा)

```
List<String> names = List.of("Aman", "Ankit", "Abhinav",
                             "Durgesh");
```

* Filter Name which Started with 'A'

यहाँपर Dimag लगाना है कि Name 'A' से Start हो रहा है कि नहीं To Boolean का कंडीशन होगा तब = filter Use करेंगे।

```
List<String> newlist = names.stream().filter(e -> e.startsWith("A")).
                       Collect(Collector.toList());
                                    // make it local
        Sop(newlist);
    }
}
```

Stream API

* // map (function)
// Each element operation

List<Integer> numbers = List.of(23, 4, 2, 5, 7, 4);

* To find the Number to get Square of each number which is present into List.

number.stream().map(i -> i*i).Collect(Collectors.toList());
↓
make it local-

Sop(new List);

}

Result = [529, 16, 4, 25, 49, 16].

}

// Sorted

* Java-1.8 (Optional class)

Optional class in Java 8 is used to avoid NullPointerException. It forces programmers to think about the case when the value is not present and take appropriate steps for Handling it.

* Or Also we can say it is a wrapper class that

- encapsulates the absence or presence of values.

Old method

```
Public class OptionalExample {
        Psvm() {                    } Common
    String str = "Java is an awsome";
     if (Str = null)
      {
        Sopln("This is null object");
     } else {
          Sopln(str.length());
      }
```

New or

```
String str = "Java is my favourite";
Ex-  if => String str = null;
Optional<String> optional = optional.ofNullable(Str);
        Sopln(optional.ispresent());
        Sopln(optional.get());          => NoSuchElement Exception in case of Str=null;
    Sopln(optional.orElse(other: " No value is this
                                Object"));
                "Java is my favourite.
    // No value is this object => in case of
                                Str=null;
}
```

Q what is time & Date format introduced into Java 8 features?

New:-

Java provides the date & time functionality with the help of two package java.time and java.util. The package java.time is introduced in Java 8. & Newly introduced classes tries to overcome the sortcomings of legacy of java.util.Date & java.util.Calender classes.

Old: Classical Date & Time API Classes

① java.lang.System

② java.util.Date. → Not thread safe.

③ java.util.Calender.

④ java.text.SimpleDateFormat.

⑤ ~~java.util.SimpleDateFormat~~ java.util.TimeZone.

⑥ Unlike old java.util.Date which is not thread safe but new date-time API is immutable & thread safe.

⑧ In old API there is only few date operations but new API provides us with many date format.

Q How is functional Interface different from a Regular interface in Java?

A functional Interface has exactly one Abstract method, while a Regular interface can have multiple Abstract methods.

Q What is predicate & function in Java Steam API.