Java 8

Before Java 8, Sun people gave importance only for objects but in 1.8 version oracle people gave the importance for functional aspects of programming to bring its benefits to Java i.e., it doesn't mean **Java** is functional oriented programming language. The main emphasis on Java 8 features is to provide support Functional Programming and Concise Code. Many New Features were introduced as a part of Java 8.

1.. Lambda Expressions

- Lambda Expressions is an anonymous function. An anonymous function is a function without name, without return type and without modifiers.
- We need -> Operator to write Lambda Expressions.
- The main advantage of Lambda Expression is to provide the support for Functional Programming in Java Programming.

Let's see how to write the Lambda Expression:

```
1  public void m1()
2  {
3      System.out.println("Hello!!");
4  }
5
```

This method can be written by using Lambda Expression as:

Main.java Run

```
1  ()-> System.out.println("Hello!!");
2
```

If Method Contains Only One line, then {} is Optional. If multiple lines are there then {} are mandatory. If we have an argument in method. Just say like in below example:

Main.java Run

```
1  public void m1(int a, int b)
2  {
3      System.out.println(a+b);
4  }
```

If the type of the parameter can be decided by compiler automatically based on the context, then we can remove types also. Above method can be written by using Lambda Expression as:

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

Let's take an example where return type is not void.

```
1  public int squareIt(int n)
2  {
3      return n*n;
4  }
```

This we can represent by using Lambda Expression as:

```
6  n -> n*n ;
7
```

Here,

- A Lambda Expression can have 0 or any number of Arguments separated by comma (,). If method has only one argument, then we can simply write as n instead of (n) i.e., parenthesis is optional. But if method has multiple argument then we cannot write as a,b we have to write it as (a,b).

  n -> n * n;
  (a,b) -> System.out.println(a+b);

  Hence, Parenthesis () is only optional if Number of Argument is 1. It is mandatory if Number of arguments is 0 or More than 1.

- If we want to use return statement explicitly then Curly Braces are mandatory.

  n -> { return n * n; }

  If we don't want to use return statement explicitly then Curly Braces are optional.

  n -> n * n;

  Hence, return is Optional while writing Lambda Expressions.

Note: Once we write lambda expression we can call that expression just like a method, for this functional interface are required.

2.. Functional Interface

An Interface which contains only method which is called **Functional Method** or **Single Abstract Method** is called **Functional Interface**. Common Examples of Functional Interfaces are: Runnable, Callable, Comparable, Comparator, ActionListener etc. If we want to invoke **Lambda Expressions**, then we need Functional Interface. Because we need the reference of Functional Interface to hold the Lambda Expression.

In Functional Interface, we should have exactly one abstract method and it can have any number of default and static methods. If we take more than one abstract method then we will get compilation error. Also if we don't have any abstract method in a Functional Interface then also we will get Compilation Error.

```
@FunctionalInterface
interface FunctionInterfaceDemo
{
}
class Demo {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

```
ERROR!
javac /tmp/2UTuQlBkn4/Demo.java
/tmp/2UTuQlBkn4/Demo.java:1: error: Unexpected @FunctionalInterface annotation
@FunctionalInterface
^
FunctionInterfaceDemo is not a functional interface
    no abstract method found in interface FunctionInterfaceDemo
1 error
```

```
@FunctionalInterface
interface FunctionInterfaceDemo
{
    public void m1();
    public void m2();
}
class Demo {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

```
ERROR!
javac /tmp/2UTuQlBkn4/Demo.java
/tmp/2UTuQlBkn4/Demo.java:1: error: Unexpected @FunctionalInterface annotation
@FunctionalInterface
^
FunctionInterfaceDemo is not a functional interface
    multiple non-overriding abstract methods found in interface FunctionInterfaceDemo
1 error
```

In Java 8, all functional Interface are annotated with optional **@FunctionalInterface** annotation which explicitly specifies that given interface is a Functional Interface or not.

2.1 Functional Interface with respect to Inheritance

If Parent Interface is a Functional Interface, then the child Interface will be Functional Interface if and only if in Child Interface either we have not declared any new abstract method because, abstract method of Parent Functional Interface will be available to the Child Interface or we have declared the same abstract method in Child Interface which is present in Parent Interface.

Hence, Below Code Snippet is Valid.

```
@FunctionalInterface
interface FunctionInterface1 {
    public void m1();
}
@FunctionalInterface
interface FunctionInterface2 extends FunctionInterface1 {
}
class Demo {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

```
java -cp /tmp/2UTuQlBkn4 Demo
Hello, World!
```

```
@FunctionalInterface
interface FunctionInterface1 {
    public void m1();
}
@FunctionalInterface
interface FunctionInterface2 extends FunctionInterface1 {
    public void m1();
}
```

```
java -cp /tmp/2UTuQlBkn4 Demo
Hello, World!
```

Hence, Below Code Snippet is Invalid.

```
@FunctionalInterface
interface FunctionInterface1 {
    public void m1();
}
@FunctionalInterface
interface FunctionInterface2 extends FunctionInterface1 {
    public void m2();
}
```

```
ERROR!
javac /tmp/2UTuQlBkn4/Demo.java
/tmp/2UTuQlBkn4/Demo.java:5: error: Unexpected @FunctionalInterface annotation
@FunctionalInterface
^
FunctionInterface2 is not a functional interface
multiple non-overriding abstract methods found in interface FunctionInterface2
1 error
```

This code will work fine if we make Child Interface as Non-Functional Interface.

```
@FunctionalInterface
interface FunctionInterface1 {
    public void m1();
}
interface FunctionInterface2 extends FunctionInterface1 {
    public void m2();
}
class Demo {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

```
java -cp /tmp/2UTuQlBkn4 Demo
Hello, World!
```

2.2 Lambda Expression Using Functional Interface

Once we write Lambda expressions to invoke its functionality, then **Functional Interface** is required. We can use Functional Interface reference to refer **Lambda Expression**. Where ever Functional Interface concept is applicable there we can use Lambda Expressions.

Consider a Functional Interface with exactly one abstract method. Lets create a DemoClass which implements the functional interface and at last call that method in a very traditional Way.

Main.java | Run | Output

```
@FunctionalInterface
interface A
{
    public void m1();
}
class DemoClass implements A
{
    public void m1()
    {
        System.out.println("Hello");
    }
}
class Test
{
    public static void main(String[] args)
    {
        DemoClass d = new DemoClass();
        d.m1();
    }
}
```

```
java -cp /tmp/gWFTCNlKBS Test
Hello
```

Now, let's rewrite this Program using Lambda Expression.

Main.java | Run | Output

```
@FunctionalInterface
interface A
{
    public void m1();
}
class Test
{
    public static void main(String[] args)
    {
        A a = () -> System.out.println("Hello");
        a.m1();
    }
}
```

```
java -cp /tmp/gWFTCNlKBS Test
Hello
```

Let’s have some more examples.

Main.java

```java
@FunctionalInterface
interface Calc
{
    public void add(int a, int b);
}
class Test
{
    public static void main(String[] args)
    {
        Calc c = (a,b) -> System.out.println(a+b);
        c.add(2,3);
        c.add(12,3);
        c.add(22,33);
        c.add(2,31);
    }
}
```

Output

```
java -cp /tmp/gWFTCNlKBS Test
5
15
55
33
```

Main.java

```java
@FunctionalInterface
interface Calc
{
    public void squareIt(int n);
}
class Test
{
    public static void main(String[] args)
    {
        Calc c = n -> System.out.println(n*n);
        c.squareIt(2);
        c.squareIt(3);
        c.squareIt(22);
        c.squareIt(15);
    }
}
```

Output

```
java -cp /tmp/gWFTCNlKBS Test
4
9
484
225
```

Note: No .class files will be generated for the Lambda Expressions.

Let’s do multi-threading by implementing Runnable Interface by using Lambda Expression.

Main.java

```java
class MultiThreadingDemo
{
    public static void main(String[] args)
    {
        Runnable r = () -> {
            for(int i=1; i <=10; i++)
            {
                System.out.println("Child Thread : " + i);
            }
        };
        Thread t = new Thread(r);
        t.start();
        for(int i=1; i <=10; i++)
        {
            System.out.println("Main Thread : " + i);
        }
    }
}
```

Output

```
java -cp /tmp/gWFTCNlKBS MultiThreadingDemo
Main Thread : 1
Main Thread : 2
Main Thread : 3
Main Thread : 4
Main Thread : 5
Main Thread : 6
Main Thread : 7
Main Thread : 8
Main Thread : 9
Main Thread : 10
Child Thread : 1
Child Thread : 2
Child Thread : 3
Child Thread : 4
Child Thread : 5
Child Thread : 6
Child Thread : 7
Child Thread : 8
Child Thread : 9
Child Thread : 10
```

Let’s write Customized Class and Use Lambda Expression there.

Main.java

```java
import java.util.*;
class Employee{
    int empId;
    String empName;
    Employee(int empId, String empName){
        this.empId = empId;
        this.empName = empName;
    }

    public String toString(){
        return this.empId + ":" + this.empName;
    }
}
class Test {
    public static void main(String[] args) {
       ArrayList<Employee> al = new ArrayList<Employee>();
       al.add(new Employee(91,"Vikash"));
       al.add(new Employee(11,"Madan"));
       al.add(new Employee(131,"Anushka"));
       al.add(new Employee(3,"Chanchal"));
       al.add(new Employee(12,"Akash"));
       al.add(new Employee(100,"Shailesh"));
       System.out.println("Before Sorting : " + al);
       Collections.sort(al,(e1,e2) -> (e1.empId < e2.empId)?-1:(e1.empId > e2.empId)?1:0);
       System.out.println("After Sorting Ascending Order: " + al);
       Collections.sort(al,(e1,e2) -> (e1.empId > e2.empId)?-1:(e1.empId < e2.empId)?1:0);
       System.out.println("After Sorting Descendin Order: " + al);
    }
}
```

Output

```
java -cp /tmp/oOhArovEqm Test
Before Sorting : [91:Vikash, 11:Madan, 131:Anushka, 3:Chanchal, 12:Akash, 100:Shailesh]
After Sorting Ascending Order: [3:Chanchal, 11:Madan, 12:Akash, 91:Vikash, 100:Shailesh, 131:Anushka]
After Sorting Descendin Order: [131:Anushka, 100:Shailesh, 91:Vikash, 12:Akash, 11:Madan, 3:Chanchal]
```

2.3 Anonymous Inner Classes vs Lambda Expressions

Anonymous Inner class	Lambda Expression
It's a class without name	It's a method without name (anonymous function)
Anonymous inner class can extend Abstract and concrete classes	lambda expression can't extend Abstract and concrete classes
Anonymous inner class can implement An interface that contains any number of Abstract methods	lambda expression can implement an Interface which contains single abstract method (Functional Interface)
Inside anonymous inner class we can Declare instance variables.	Inside lambda expression we can't Declare instance variables, whatever the variables declared are simply acts as local variables.
Anonymous inner classes can be Instantiated	lambda expressions can't be instantiated
Inside anonymous inner class "this" Always refers current anonymous Inner class object but not outer class Object.	Inside lambda expression "this" Always refers current outer class object. That is enclosing class object.
Anonymous inner class is the best choice If we want to handle multiple methods.	Lambda expression is the best Choice if we want to handle interface With single abstract method (Functional Interface).
In the case of anonymous inner class At the time of compilation a separate Dot class file will be generated (outerclass$1.class)	At the time of compilation no dot Class file will be generated for Lambda expression. It simply converts in to private method outer class.
Memory allocated on demand Whenever we are creating an object	Reside in permanent memory of JVM (Method Area).

Wherever we are using anonymous inner classes there may be a chance of using Lambda expression to reduce length of the code and to resolve complexity.

```
1)  class Test {
2)      public static void main(String[] args) {
3)          Thread t = new Thread(new Runnable() {
4)                  public void run() {
5)                      for(int i=0; i<10; i++) {
6)                          System.out.println("Child Thread");
7)                      }
8)                  }
9)          });
10)         t.start();
11)         for(int i=0; i<10; i++)
12)             System.out.println("Main thread");
13)     }
14) }
```

```
1)  class Test {
2)      public static void main(String[] args) {
3)              Thread t = new Thread(() → {
4)                                          for(int i=0; i<10; i++) {
5)                                              System.out.println("Child Thread");
6)                                          }
7)              });
8)              t.start();
9)              for(int i=0; i<10; i++) {
10)                 System.out.println("Main Thread");
11)             }
12)     }
13) }
```

2.4 Advantages of Lambda Expressions

- We can reduce length of the code so that readability of the code will be improved.
- We can resolve complexity of anonymous inner classes.
- We can provide Lambda expression in the place of object.
- We can pass lambda expression as argument to methods.

Note:

- Anonymous inner class can extend concrete class, can extend abstract class, can implement interface with any number of methods but
- Lambda expression can implement an interface with only single abstract method (Functional Interface).
- Hence if anonymous inner class implements Functional Interface in that particular case only, we can replace with lambda expressions. Hence wherever anonymous inner class concept is there, it may not possible to replace with Lambda expressions.
- Anonymous inner class! = Lambda Expression
- Inside anonymous inner class we can declare instance variables.
- Inside anonymous inner class "this" always refers current inner class object(anonymous inner class) but not related outer class object.

Example:

- Inside lambda expression we can't declare instance variables.
- Whatever the variables declare inside lambda expression are simply acts as local variables
- Within lambda expression 'this" keyword represents current outer class object reference (that is current enclosing class reference in which we declare lambda expression).
- From lambda expression we can access enclosing class variables and enclosing method variables directly.

- The local variables referenced from lambda expression are implicitly final and hence we can't perform re-assignment for those local variables otherwise we get compile time error.

2.5 Predefined Functional Interfaces

There are various Predefined Functional Interfaces in Java:

- Predicate
- Function
- Consumer
- Supplier

Certain Two Arguments Predefined Functional Interfaces are:

- BiPredicate
- BiFunction
- BiConsumer

Certain Primitive Functional Interfaces are:

- IntPredicate, LongPredicate, DoublePredicate
- IntFunction, LongFunction, DoubleFunction
- IntConsumer, LongConsumer, DoubleConsumer

2.6 Predicate

- A predicate is a function with a single argument and returns boolean value.
- To implement predicate functions in Java, Oracle people introduced Predicate Functional Interface in Java 1.8 version.
- Predicate Functional Interface present in Java.util.function package.
- It's a functional interface and it contains single abstract method method i.e., test() for checking any condition on the Input of Type T and returns a boolean value.

```
3  interface Predicate<T>
4  {
5      public boolean test(T t);
6  }
7
```

Main.java | Run | Output

```
1  import java.util.function.*;
2  class PredicateDemo
3  {
4      public static void main(String[] args)
5      {
6          Predicate<Integer> p = i -> i % 2 == 0;
7          System.out.println(p.test(10));
8          System.out.println(p.test(11));
9          System.out.println(p.test(34));
10         System.out.println(p.test(39));
11     }
12 }
```

```
java -cp /tmp/S2PG1RRXlm PredicateDemo
true
false
true
false
```

Predicate Joining

It's possible to join predicates into a single predicate by using the following methods and these are exactly same as logical AND, OR and Complement operators

- and()
- or()
- negate()

```java
import java.util.function.*;
class PrdeicateDemo {
    public static void m1(Predicate<Integer> p, int[] x) {
        for(int x1:x) {
            if(p.test(x1))
               System.out.println(x1);
        }
    }
    public static void main(String[] args) {
        int[] x = {0, 5, 10, 15, 20, 25, 30};
        Predicate<Integer> p1 = i -> i > 10;
        Predicate<Integer> p2 = i -> i % 2 == 0;
        System.out.println("The Numbers Greater Than 10 are : ");
        m1(p1,x);
        System.out.println("The Even Numbers are : ");
        m1(p2,x);
        System.out.println("The Numbers Not Greater Than 10:");
        m1(p1.negate(), x);
        System.out.println("The Numbers Greater Than 10 And Even Are : ");
        m1(p1.and(p2), x);
        System.out.println("The Numbers Greater Than 10 OR Even : ");
        m1(p1.or(p2), x);
    }
}
```

```
java -cp /tmp/2UTuQlBkn4 PrdeicateDemo
The Numbers Greater Than 10 are :
15
20
25
30
The Even Numbers are :
0
10
20
30
The Numbers Not Greater Than 10:
0
5
10
The Numbers Greater Than 10 And Even Are :
20
30
The Numbers Greater Than 10 OR Even :
0
10
15
20
25
30
```

2.7 Function

- Functions are exactly same as Predicates except that functions can return any type of result but function should (can) return only one value and that value can be any type as per our requirement.
- To implement functions oracle people introduced Function interface in 1.8 Version.
- Function interface present in Java.util.function package.
- Functional interface contains single abstract method i.e., apply() Which will apply some business logic on the Input of Type T of method and Returns some Output of Type R.

```java
interface Function<T, R>
{
    public R apply(T t);
}
```

Main.java | Run | Output

```java
import java.util.function.*;

class FunctionDemo
{
    public static void main(String[] args)
    {
        Function<Integer, Integer> f = i -> i * i;
        System.out.println(f.apply(5));
        System.out.println(f.apply(8));
        System.out.println(f.apply(16));
        System.out.println(f.apply(21));

        Function<String, Integer> fs = s -> s.length();
        System.out.println(fs.apply("Vikash"));
        System.out.println(fs.apply("Tirupati"));
    }
}
```

```
java -cp /tmp/S2PGiRRXlm FunctionDemo
25
64
256
441
6
8
```

Function Chaining

We can do function Chaing as Follows:

F1.andThen(F2) -> First F1 and then F2 is called.

F1.compose(F2) -> First F2 and then F1 is called.

Main.java

Run

Output

```
import java.util.function.*;

class FunctionDemo {
    public static void main(String[] args) {
        Function<Integer,Integer> f1 = i -> 2*i;
        Function<Integer,Integer> f2 = i -> i*i*i;
        System.out.println(f1.andThen(f2).apply(2));
        System.out.println(f1.compose(f2).apply(2));
    }
}
```

```
java -cp /tmp/efHypSxQVx FunctionDemo
64
16
```

2.8 Difference between Function and Predicate

Predicate	Function
To implement conditional checks We should go for predicate	To perform certain operation And to return some result we Should go for function.
Predicate can take one type Parameter which represents Input argument type. Predicate<T>	Function can take 2 type Parameters. First one represent Input argument type and Second one represent return Type. Function<T,R>
Predicate interface defines only one method called test()	Function interface defines only one Method called apply().
public boolean test(T t)	public R apply(T t)
Predicate can return only boolean value.	Function can return any type of value

2.9 Consumer

Consumer is a Functional Interface which provides a single abstract method accept() where it takes an Input of Type T and perform some operation and won't return anything.

```
interface Consumer<T>
{
    public void accept(T t);
}
```

Main.java

Run

Output

```
import java.util.function.*;

class ConsumerDemo {
    public static void main(String[] args) {
        Consumer<String> c = i -> System.out.println(i.length());
        c.accept("Vikash");
        c.accept("Naina");
        c.accept("Naitik");
        c.accept("Navika");
    }
}
```

```
java -cp /tmp/efHypSxQVx ConsumerDemo
6
5
6
6
```

2.10 Supplier

Supplier is a Functional Interface which provides a single abstract method get() where it doesn't take any input but just supply a required Objects with return type R.

```
interface Supplier<R>  // R is a Return Type not an input Type
{
    public R get();
}
```

Main.java

```
import java.util.function.*;
import java.util.*;

class SupplierDemo {
    public static void main(String[] args) {
        Supplier<Date> s = () -> new Date();
        System.out.println(s.get());
    }
}
```

Run

Output

```
java -cp /tmp/efHypSxQVx SupplierDemo
Sun Jun 18 16:55:24 GMT 2023
```

2.11 BiPredicate

It is similar to Predicate, only difference is it will take two input argument.

```
interface BiPredicate<T1,T2>
{
    public boolean test(T1 t1,T2 t2);
}
```

Main.java

```
import java.util.function.*;

class BiPredicateDemo {
    public static void main(String[] args) {
        BiPredicate<Integer, Integer> p = (a,b) -> (a+b) % 2 == 0;
        System.out.println(p.test(2,3));
        System.out.println(p.test(3,3));
    }
}
```

Run

Output

```
java -cp /tmp/efHypSxQVx BiPredicateDemo
false
true
```

2.12 BiFunction

It is similar to Function, only difference is it will take two input argument.

```
interface BiFunction<T,U,R>
{
    public R apply(T t, U u);
}
```

Main.java

```
import java.util.function.*;

class BiFunctionDemo {
    public static void main(String[] args) {
        BiFunction<Integer, Integer, Integer> f = (a,b) -> (a-b) * (a-b);
        System.out.println(f.apply(2,3));
        System.out.println(f.apply(3,3));
    }
}
```

Run

Output

```
java -cp /tmp/efHypSxQVx BiFunctionDemo
25
36
```

2.13 BiConsumer

It is similar to Consumer, only difference is it will take two input argument.

```
interface BiConsumer<T,U>
{
    public void accept(T t, U u);
}
```

Main.java

Run

Output

```
import java.util.function.*;

class BiConsumerDemo {
    public static void main(String[] args) {
        BiConsumer<String, String> f = (a,b) -> System.out.println(a.length()+b.length());
        f.accept("Vikash","Sharma");
    }
}
```

```
java -cp /tmp/efHypSxQVx BiConsumerDemo
12
```

3.. Default Methods

Until 1.7 version onwards inside interface we can take only public abstract methods and public static final variables (every method present inside interface is always public and abstract whether we are declaring or not) and Every variable declared inside interface is always public static final whether we are declaring or not.

But from 1.8 version onwards in addition to these, we can declare **Default** concrete methods also inside interface, which are also known as Defender methods or **Virtual Extension Method**.

The main advantage of using this kind of methods is that without effecting the implementation classes if we want to add any new method to the interface then we can go and create a default method in the interface.

Default method is created with '**default'** keyword and we provide the default implementation of that method within an Interface itself. If any implementation class wants to use this default method, they can use it or if any class wants to override the implementation for default method, they can do it without any issue.


```
interface DefaultInterface
{
    public void m1();
    default void m2()
    {
        System.out.println("Default Implementation of m2");
    }
}
class Demo implements DefaultInterface
{
    public void m1()
    {
        System.out.println("Implementation of m1");
    }
}
class Test
{
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.m1();
        d.m2();
    }
}
```

Here., we have an interface named "DefaultInterface". In this interface we have one abstract method m1(). In class Demo we have provided the implementation of m1() method. In Interface we have also declared one default method m2() with default implementation which will be by default available to Demo Class and we can call it via Demo class Object.

If we are not happy with m2() method default implementation then we can happily override it in implementation class. While overriding it in implementation class default will be replaced with public as we cannot have default method in class. If we declare default method in class then it will throw compile time error. See the below program.

Main.java

Run

Output

```
interface DefaultInterface
{
    public void m1();
    default void m2()
    {
        System.out.println("Default Implementation of m2");
    }
}
class Demo implements DefaultInterface
{
    public void m1()
    {
        System.out.println("Implementation of m1");
    }
    public void m2()
    {
        System.out.println("Overridden Implementation of m2");
    }
}
class Test
{
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.m1();
        d.m2();
    }
}
```

```
java -cp /tmp/p2erS6AXXw Test
Implementation of m1
Overridden Implementation of m2
```

Note: We cannot use any methods of Object class as a default method in Interface this is because implementation class is by default the child of Object class so Object class method will be by default available to Implementation Class.

3.1 Default methods Versus Multiple Inheritance

Two interfaces can contain default method with same signature then there may be a chance of ambiguity problem (Diamond problem) to the implementation class. To overcome this problem compulsory, we should override default method in the implementation class otherwise we get compile time error.

```
import java.util.function.*;
interface Left {
    default void m1() {
        System.out.println("Left Class m1");
    }
}
interface Right {
    default void m1() {
        System.out.println("Right Class m1");
    }
}
class Test implements Left, Right {

}
class DefaultDemo {
    public static void main(String[] args) {
    }
}
```

```
ERROR!
javac /tmp/2UTuQlBkn4/DefaultDemo.java
/tmp/2UTuQlBkn4/DefaultDemo.java:12: error: types Left and Right are incompatible;
class Test implements Left, Right {
^
  class Test inherits unrelated defaults for m1() from types Left and Right
1 error
```

```java
import java.util.function.*;
interface Left {
    default void m1() {
        System.out.println("Left Class m1");
    }
}
interface Right {
    default void m1() {
        System.out.println("Right Class m1");
    }
}
class Test implements Left, Right {
    public void m1(){
        System.out.println("Test Class m1");
    }
}
class DefaultDemo {
    public static void main(String[] args) {
    }
}
```

```
java -cp /tmp/2UTuQlBkn4 DefaultDemo
```

3.2 Default Methods versus Abstract Class

Even though we can add concrete methods in the form of default methods to the interface, it won't be equal to abstract class.

Interface with Default Methods	Abstract Class
Inside interface every variable is Always public static final and there is No chance of instance variables	Inside abstract class there may be a Chance of instance variables which Are required to the child class.
Interface never talks about state of Object.	Abstract class can talk about state of Object.
Inside interface we can't declare Constructors.	Inside abstract class we can declare Constructors.
Inside interface we can't declare Instance and static blocks.	Inside abstract class we can declare Instance and static blocks.
Functional interface with default Methods Can refer lambda expression.	Abstract class can't refer lambda Expressions.
Inside interface we can't override Object class methods.	Inside abstract class we can override Object class methods.

4.. Static Methods inside Interface

From 1.8 version onwards in addition to default methods we can write static methods also inside interface to define utility functions. Interface static methods by-default not available to the implementation classes hence by using implementation class reference we can't call interface static methods. We should call interface static methods by using interface name.

```
interface Interf {
    public static void sum(int a, int b) {
        System.out.println("Sum is : " + (a+b));
    }
}
class StaticDemo implements Interf {
    public static void main(String[] args) {
        Interf.sum(10,20);
    }
}
```

```
java -cp /tmp/2UTuQlBkn4 StaticDemo
Sum is : 30
```

```
interface Interf {
    public static void sum(int a, int b) {
        System.out.println("Sum is : " + (a+b));
    }
}
class StaticDemo implements Interf {
    public static void main(String[] args) {
        StaticDemo demo = new StaticDemo();
        demo.sum(10,20);
    }
}
```

```
ERROR!
javac /tmp/2UTuQlBkn4/StaticDemo.java
/tmp/2UTuQlBkn4/StaticDemo.java:9: error: cannot find symbol
demo.sum(10,20);
         ^
  symbol:   method sum(int,int)
  location: variable demo of type StaticDemo
1 error
```

```
interface Interf {
    public static void sum(int a, int b) {
        System.out.println("Sum is : " + (a+b));
    }
}
class StaticDemo implements Interf {
    public static void main(String[] args) {
        StaticDemo.sum(10,20);
    }
}
```

```
ERROR!
javac /tmp/2UTuQlBkn4/StaticDemo.java
/tmp/2UTuQlBkn4/StaticDemo.java:8: error: cannot find symbol
        StaticDemo.sum(10,20);
                   ^
  symbol:   method sum(int,int)
location: class StaticDemo
1 error
```

As interface static methods by default not available to the implementation class, overriding concept is not applicable. Based on our requirement we can define exactly same method in the implementation class, it's valid but not overriding.

```
1) interface Interf {
2)         public static void m1() {}
3) }
4) class Test implements Interf {
5)           public static void m1() {}
6) }
```

But it is not Method Overriding.

From 1.8 version onwards we can write main() method inside interface and hence we can run interface directly from the command prompt.

```
1) interface Interf {
2)         public static void main(String[] args) {
3)               System.out.println("Interface Main Method");
4)         }
5) }
```

At the command prompt:
Javac Interf.Java
JavaInterf

5.. Method References / Double Colon (::) Operator

Java provides a new feature called method reference in Java 8. Functional Interface method can be mapped to our specified method by using :: (double colon) operator. This is called **Method Reference**. Our specified method can be either static method or instance method. Functional Interface method and our specified method should have same argument types, except this the remaining things like return type, method name, modifiers etc. are not required to match.

Functional Interface can refer lambda expression and Functional Interface can also refer method reference. Hence lambda expression can be replaced with method reference. Hence method reference is alternative syntax to lambda expression.

Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

There are Following types of Method References in Java:

- Reference to a Static Method
- Reference to an Instance Method
- Reference to a Constructor

5.1 Reference to a Static Method

We can refer static methods defined inside a class. The general form Syntax of Referencing Static Method is:

<Class Name Containing Method>::<Method Name>

Example Code Snippet:

Main.java

```
interface Sayable{
    void say();
}
public class MethodReference
{
    public static void saySomething()
    {
        System.out.println("Hello, this is static method.");
    }
    public static void main(String[] args)
    {
        Sayable sayable = MethodReference::saySomething;
        sayable.say();
    }
}
```

Output

```
java -cp /tmp/OzwQO1qOUv MethodReference
Hello, this is static method.
```

Let's try Multithreading using Static Method Reference.

Main.java

```
class Test {
    public static void m1()
    {
        for(int i=1; i <= 10; i++)
        {
            System.out.println("Child Thread : " + i);
        }
    }
    public static void main(String[] args)
    {
        Runnable r = Test::m1;
        Thread t = new Thread(r);
        t.start();
        for(int i=1; i <= 10; i++)
        {
            System.out.println("Main Thread : " + i);
        }
    }
}
```

Output

```
java -cp /tmp/OzwQO1qOUv Test
Child Thread : 1
Child Thread : 2
Child Thread : 3
Child Thread : 4
Main Thread : 1
Main Thread : 2
Main Thread : 3
Main Thread : 4
Main Thread : 5
Main Thread : 6
Main Thread : 7
Main Thread : 8
Main Thread : 9
Main Thread : 10
Child Thread : 5
Child Thread : 6
Child Thread : 7
Child Thread : 8
Child Thread : 9
Child Thread : 10
```

Here m1() method which is a static method is referred to as a run() method of Runnable Functional Interface.

5.2 Reference to an Instance Method

We can refer instance methods defined inside a class. The general form Syntax of Referencing Instance Method is:

<Object Reference>::<Method Name>

Example Code Snippet

Main.java

Run

Output

```
interface Sayable{
    void say();
}
public class InstanceMethodReference {
    public void saySomething(){
        System.out.println("Hello, this is non-static method.");
    }
    public static void main(String[] args)
    {
        InstanceMethodReference methodReference = new InstanceMethodReference();
        Sayable sayable = methodReference::saySomething;
        sayable.say();
    }
}
```

```
java -cp /tmp/OzwQO1qOUv InstanceMethodReference
Hello, this is non-static method.
```

5.3 Reference to a Constructor

We can refer constructor of a class. The general form Syntax of Referencing Constructor is:

<Class Name>::new

Main.java

Run

```
interface Messageable
{
    Message getMessage(String msg);
}

class Message
{
    Message(String msg)
    {
        System.out.print(msg);
    }
}
public class ConstructorReference
{
    public static void main(String[] args)
    {
        Messageable hello = Message::new;
        hello.getMessage("Hello");
    }
}
```

6.. Streams API

If we want to represent a group of Objects as a single entity, then we should go for Collection. If we want to process the Objects of Collection, then we should go for Stream concept.

What is the differences between Java.util.streams and Java.io streams?

java.util streams meant for processing objects from the collection i.e., it represents a stream of objects from the collection but Java.io streams meant for processing binary and character data with respect to file i.e., it represents stream of binary data or character data from the file. Hence, Java.io streams and Java.util streams both are different.

What is the difference between collection and stream?

If we want to represent a group of individual objects as a single entity then We should go for collection. If we want to process a group of objects from the collection then we should go for streams. We can create a stream object to the collection by using stream() method of Collection interface. stream() method is a default method added to the Collection in 1.8 version.

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

Stream s = c.stream();

Here c is any collection or Map type Object.

We can process the objects in the following 2 phases

- Configuration
- Processing

6.1 Configuration

Configuration can be done by using either of the below Mechanisms:

- Filter
- Map
- FlatMap

6.1.1 Filter

Filter is used to filter out the elements from the Collection based on some filtering criteria. In filter() method, we pass Predicate as an argument. filter(Predicate P);

Main.java

Run

Output

```
import java.util.*;
import java.util.stream.*;

class FilterDemo
{
    public static void main(String[] args)
    {
       ArrayList<Integer> al = new ArrayList<Integer>();
       al.add(20);
       al.add(21);
       al.add(22);
       al.add(23);
       al.add(19);
       al.add(17);
       al.add(24);
       al.add(26);
       System.out.println(al);
       List<Integer> l = al.stream().filter(i -> i % 2 == 0).collect(Collectors.toList());
       System.out.println(l);
    }
}
```

```
java -cp /tmp/OzwQO1qOUv FilterDemo

[20, 21, 22, 23, 19, 17, 24, 26]
[20, 22, 24, 26]
```


```java
import java.util.*;
import java.util.stream.*;

class FilterDemo
{
    public static void main(String[] args)
    {
        ArrayList<Integer> al = new ArrayList<Integer>();
        al.add(20);
        al.add(21);
        al.add(22);
        al.add(23);
        al.add(19);
        al.add(17);
        al.add(24);
        al.add(26);
        System.out.println(al);
        List<Integer> l = al.stream().filter(i -> i % 2 != 0).collect(Collectors.toList());
        System.out.println(l);
    }
}
```

6.1.2 Map

If we want to create a separate new object, for every object present in the collection based on our requirement then we should go for map() method of Stream interface. Map is used to perform an operation on every element of the collection. In map() method, we pass Function as an argument. map(Function F);


```java
import java.util.*;
import java.util.stream.*;

class MapDemo
{
    public static void main(String[] args)
    {
        ArrayList<Integer> al = new ArrayList<Integer>();
        al.add(20);
        al.add(21);
        al.add(22);
        al.add(23);
        al.add(19);
        al.add(17);
        al.add(24);
        al.add(26);
        System.out.println(al);
        List<Integer> l = al.stream().map(i -> i * i).collect(Collectors.toList());
        System.out.println(l);
    }
}
```

6.1.3 FlatMap

In Java 8 Streams, the flatMap() method applies operation as a mapper function and provides a stream of element values. It means that in each iteration of each element the map() method creates a separate new stream. By using the flattening mechanism, it merges all streams into a single resultant stream. In short, it is used to convert a Stream of Stream into a list of values.

In short, we can say that the flatMap() method helps in converting Stream<Stream<T>> to Stream<T>. It performs flattening (flat or flatten) and mapping (map), simultaneously. The Stream.flatMap() method combines both the operations i.e. flat and map. **Flattening** is the process of converting several lists of lists and merge all those lists to create a single list containing all the elements from all the lists.

Main.java

```java
import java.util.*;
import java.util.stream.*;
import java.util.function.*;
class ForEachDemo{
    public static void main(String[] args) {
        List<String> productlist1 = Arrays.asList("Printer", "Mouse", "Keyboard", "Motherboard");
        List<String> productlist2 = Arrays.asList("Scanner", "Projector", "Light Pen");
        List<String> productlist3 = Arrays.asList("Pen Drive", "Charger", "WIFI Adapter", "Cooling
            Fan");
        List<String> productlist4 = Arrays.asList("CPU Cabinet", "WebCam", "USB Light", "Microphone",
            "Power cable");
        List<List<String>> allproducts = new ArrayList<List<String>>();
        allproducts.add(productlist1);
        allproducts.add(productlist2);
        allproducts.add(productlist3);
        allproducts.add(productlist4);
        List<String> listOfAllProducts = new ArrayList<String>();
        for(List<String> pro : allproducts){
            for(String product : pro){
                listofallProducts.add(product);
            }
        }
        System.out.println("List Before Applying mapping and Flattening: \n");
        System.out.println(listOfAllProducts);
        System.out.println();
        List<String> flatMapList = allproducts .stream().flatMap(plist -> plist.stream()).collect
            (Collectors.toList());
        System.out.println("List After Applying Mapping and Flattening Operation: \n");
        System.out.println(flatMapList);
    }
}
```

Output

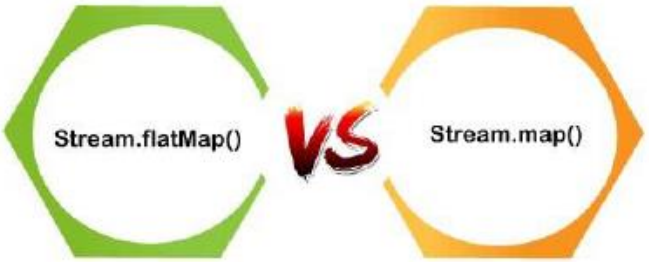
```
java -cp /tmp/2ByGrBCPkT ForEachDemo
List Before Applying mapping and Flattening:

[Printer, Mouse, Keyboard, Motherboard, Scanner, Projector, Light Pen, Pen Drive, Charger, WIFI Adapter,
    Cooling Fan, CPU Cabinet, WebCam, USB Light, Microphone, Power cable]

List After Applying Mapping and Flattening Operation:

[Printer, Mouse, Keyboard, Motherboard, Scanner, Projector, Light Pen, Pen Drive, Charger, WIFI Adapter,
    Cooling Fan, CPU Cabinet, WebCam, USB Light, Microphone, Power cable]
```

The following table describes the key differences between Stream.flatMap() and Stream.map().



Stream.flatMap()	Stream.map()
It processes the stream of stream's values.	It processes the stream of values.
It performs mapping along with flattening.	It performs mapping only.
It transforms data from Stream> to Stream.	It transforms data from Stream to Stream.
It uses One-To-Many mapping.	It uses One-To-One mapping.
It's mapper function produces multiple values (stream of values) for each input value.	It's mapper function produces single values for each input value.
Use the flatMap() method when the mapper function is producing multiple values for each input value.	Use the map() method when the mapper function is producing single values for each input value.

6.2 Processing

We can do various Processing of the objects present in the Collection. This Processing Includes:

- collect() Method
- count() Method
- sorted() Method
- min() and max() Method
- forEach() Method
- toArray() Method
- Stream.of() Method

6.2.1 collect() Method

This method collects the elements from the Stream and adding it to the specified Collection indicated (specified) by argument.

Example:

To Collect Only Even Numbers from the ArrayList.

```
1)  import Java.util.*;
2)  class Test {
3)          public static void main(String[] args) {
4)                  ArrayList<Integer> l1 = new ArrayList<Integer>();
5)                  for(int i=0; i<=10; i++) {
6)                      l1.add(i);
7)                  }
8)                  System.out.println(l1);
9)                  ArrayList<Integer> l2 = new ArrayList<Integer>();
10)                 for(Integer i:l1) {
11)                         if(i%2 == 0)
12)                             l2.add(i);
13)                 }
14)                 System.out.println(l2);
15)         }
16) }
```

```
1)  import Java.util.*;
2)  import Java.util.stream.*;
3)  class Test {
4)          public static void main(String[] args) {
5)                  ArrayList<Integer> l1 = new ArrayList<Integer>();
6)                  for(inti=0; i<=10; i++) {
7)                          l1.add(i);
8)                  }
9)                  System.out.println(l1);
10)                 List<Integer> l2 = l1.stream().filter(i -> i%2==0).collect(Collectors.toList());
11)                 System.out.println(l2);
12)         }
13) }
```

```
1)  import Java.util.*;
2)  import Java.util.stream.*;
3)  class Test {
4)          public static void main(String[] args) {
5)                  ArrayList<String> l = new ArrayList<String>();
6)                  l.add("rvk"); l.add("rk"); l.add("rkv"); l.add("rvki"); l.add("rvkir");
7)                  System.out.println(l);
8)                  List<String> l2 = l.Stream().map(s -
    >s.toUpperCase()).collect(Collectors.toList());
9)                  System.out.println(l2);
10)         }
11) }
```

6.2.2 count() Method

Count method is used to count the number of elements in a stream i.e., in the processed Collection and return the count of long type.

public long count()

Main.java

```
import java.util.*;
import java.util.stream.*;

class CountDemo {
    public static void main(String[] args) {
        ArrayList<Integer> al = new ArrayList<Integer>();
        al.add(20);
        al.add(21);
        al.add(22);
        al.add(23);
        al.add(19);
        al.add(17);
        al.add(24);
        al.add(26);
        System.out.println(al);
        long l = al.stream().filter(i -> i < 20).count();
        System.out.println(l);
    }
}
```

Output

```
java -cp /tmp/2ByGrRCPkT CountDemo
[20, 21, 22, 23, 19, 17, 24, 26]
2
```

6.2.3 sorted() Method

sorted() method is used to sort the stream. If we want to sort the elements present inside stream then we should go for sorted() method and the sorting can either default natural sorting order or customized sorting order specified by comparator.

sorted() - Default natural sorting order

sorted(Comparator c) - Customized sorting order.

Main.java

```
import java.util.*;
import java.util.stream.*;

class SortedDemo {
    public static void main(String[] args) {
        ArrayList<Integer> al = new ArrayList<Integer>();
        al.add(20);
        al.add(21);
        al.add(22);
        al.add(23);
        al.add(19);
        al.add(17);
        al.add(24);
        al.add(26);
        System.out.println(al);
        List<Integer> l = al.stream().sorted().collect(Collectors.toList());
        System.out.println(l);
    }
}
```

Output

```
java -cp /tmp/2ByGrRCPkT SortedDemo
[20, 21, 22, 23, 19, 17, 24, 26]
[17, 19, 20, 21, 22, 23, 24, 26]
```

Main.java

Run

Output

```
import java.util.*;
import java.util.stream.*;

class SortedDemo {
    public static void main(String[] args) {
        ArrayList<Integer> al = new ArrayList<Integer>();
        al.add(20);
        al.add(21);
        al.add(22);
        al.add(23);
        al.add(19);
        al.add(17);
        al.add(24);
        al.add(26);
        System.out.println(al);
        List<Integer> l = al.stream().sorted((a,b) -> b.compareTo(a)).collect(Collectors.toList());
        System.out.println(l);
    }
}
```

```
java -cp /tmp/2ByGrRCPkT SortedDemo
[20, 21, 22, 23, 19, 17, 24, 26]
[26, 24, 23, 22, 21, 20, 19, 17]
```

Sort List of String objects in Ascending order using Java 8 Stream APIs

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;
public class StreamListSorting {
    public static void main(String[] args) {
        List < String > fruits = new ArrayList < String > ();
        fruits.add("Banana");
        fruits.add("Apple");
        fruits.add("Mango");
        fruits.add("Orange");
        List<String> sortedList = fruits.stream().sorted(Comparator.naturalOrder()).collect(Collectors
            .toList());
        System.out.println(sortedList);
        List < String > sortedList1 = fruits.stream().sorted((o1, o2) -> o1.compareTo(o2)).collect
            (Collectors.toList());
        System.out.println(sortedList1);
        List < String > sortedList2 = fruits.stream().sorted().collect(Collectors.toList());
        System.out.println(sortedList2);
    }
}
```

```
java -cp /tmp/xfsBMpLDbi StreamListSorting
[Apple, Banana, Mango, Orange]
[Apple, Banana, Mango, Orange]
[Apple, Banana, Mango, Orange]
```

Sort List of String objects in Descending order using Java 8 Stream APIs

```
import java.util.Comparator;
import java.util.*;
import java.util.stream.Collectors;
public class StreamListSorting {
    public static void main(String[] args) {
        List < String > fruits = new ArrayList < String > ();
        fruits.add("Banana");
        fruits.add("Apple");
        fruits.add("Mango");
        fruits.add("Orange");
        // descending order
        List <String> sortedList3 = fruits.stream().sorted(Comparator.reverseOrder()).collect
            (Collectors.toList());
        System.out.println(sortedList3);
        List <String> sortedList4 = fruits.stream().sorted((o1, o2) -> o2.compareTo(o1)).collect
            (Collectors.toList());
        System.out.println(sortedList4);
    }
}
```

```
java -cp /tmp/xfsBMpLDbi StreamListSorting
[Orange, Mango, Banana, Apple]
[Orange, Mango, Banana, Apple]
```

Sort List of Employee objects in Ascending order using Java 8 Stream APIs

```java
class Employee {
    private int id;
    private String name;
    private int age;
    private long salary;
    public Employee(int id, String name, int age, long salary) {
        super();
        this.id = id;
        this.name = name;
        this.age = age;
        this.salary = salary;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public long getSalary() {
        return salary;
    }
    public void setSalary(long salary) {
        this.salary = salary;
    }
    @Override
    public String toString() {
        return "Employee [id=" + id + ", name=" + name + ", age=" + age + ", salary=" + salary + "]";
    }
}
class StreamListSorting {
    public static void main(String[] args) {
        List < Employee > employees = new ArrayList < Employee > ();
        employees.add(new Employee(10, "Ramesh", 30, 400000));
        employees.add(new Employee(20, "John", 29, 350000));
        employees.add(new Employee(30, "Tom", 30, 450000));
        employees.add(new Employee(40, "Pramod", 29, 500000));

        List < Employee > employeesSortedList1 = employees.stream()
            .sorted((o1, o2) -> (int)(o1.getSalary() - o2.getSalary())).collect(Collectors.toList());
        System.out.println(employeesSortedList1);

        List < Employee > employeesSortedList2 = employees.stream()
            .sorted(Comparator.comparingLong(Employee::getSalary)).collect(Collectors.toList());
                //ascending order
        System.out.println(employeesSortedList2);
    }
}
```

```
java -cp /tmp/xfsBMpLDbi StreamListSorting
[Employee [id=20, name=John, age=29, salary=350000], Employee [id=10, name=Ramesh, age=30, salary=400000],
    Employee [id=30, name=Tom, age=30, salary=450000], Employee [id=40, name=Pramod, age=29, salary=500000]]
[Employee [id=20, name=John, age=29, salary=350000], Employee [id=10, name=Ramesh, age=30, salary=400000],
    Employee [id=30, name=Tom, age=30, salary=450000], Employee [id=40, name=Pramod, age=29, salary=500000]]
```

Sort List of Employee objects in Descending order using Java 8 Stream APIs

```
45 class StreamListSorting {
46     public static void main(String[] args) {
47         List < Employee > employees = new ArrayList < Employee > ();
48         employees.add(new Employee(10, "Ramesh", 30, 400000));
49         employees.add(new Employee(20, "John", 29, 350000));
50         employees.add(new Employee(30, "Tom", 30, 450000));
51         employees.add(new Employee(40, "Pramod", 29, 500000));
52
53         List < Employee > employeesSortedList1 = employees.stream()
54             .sorted((o1, o2) -> (int)(o2.getSalary() - o1.getSalary())).collect(Collectors.toList());
55         System.out.println(employeesSortedList1);
56
57         List < Employee > employeesSortedList2 = employees.stream()
58             .sorted(Comparator.comparingLong(Employee::getSalary).reversed()).collect(Collectors
                    .toList());
59         System.out.println(employeesSortedList2);
60     }
61 }
```

```
java -cp /tmp/xfsBMpLDbi StreamListSorting
[Employee [id=40, name=Pramod, age=29, salary=500000], Employee [id=30, name=Tom, age=30, salary=450000],
    Employee [id=10, name=Ramesh, age=30, salary=400000], Employee [id=20, name=John, age=29, salary=350000]]
[Employee [id=40, name=Pramod, age=29, salary=500000], Employee [id=30, name=Tom, age=30, salary=450000],
    Employee [id=10, name=Ramesh, age=30, salary=400000], Employee [id=20, name=John, age=29, salary=350000]]
```

6.2.4 min() and max() Methods

Min is the first element of the Sorted stream and max is the last element of the sorted stream. It doesn't depend on value. It depends on Sorting Order in which it is sorted.

Main.java | Run | Output

```
import java.util.*;
import java.util.stream.*;

class SortedDemo {
    public static void main(String[] args) {
        ArrayList<Integer> al = new ArrayList<Integer>();
        al.add(20);
        al.add(21);
        al.add(22);
        al.add(23);
        al.add(19);
        al.add(17);
        al.add(24);
        al.add(26);
        System.out.println("When Ascending Order");
        Integer min1 = al.stream().min((a,b) -> a.compareTo(b)).get();
        Integer max1 = al.stream().max((a,b) -> a.compareTo(b)).get();
        System.out.println(min1);
        System.out.println(max1);

        System.out.println("When Descending Order");
        Integer min2 = al.stream().min((a,b) -> b.compareTo(a)).get();
        Integer max2 = al.stream().max((a,b) -> b.compareTo(a)).get();
        System.out.println(min2);
        System.out.println(max2);
    }
}
```

```
java -cp /tmp/2ByGrRCPkT SortedDemo
When Ascending Order
17
26
When Descending Order
26
17
```

6.2.5 forEach() Method

For each contains Consumer Argument. It is an alternative to foreach loop while traversing Collections in Java.

Main.java

Run

```
import java.util.*;
import java.util.stream.*;
import java.util.function.*;

class ForEachDemo
{
    public static void main(String[] args)
    {
        ArrayList<Integer> al = new ArrayList<Integer>();
        al.add(20);
        al.add(21);
        al.add(22);
        al.add(23);
        al.add(19);
        al.add(17);
        al.add(24);
        al.add(26);
        Consumer<Integer> c = i -> System.out.println("Square of " + i + " is " + i*i);
        al.stream().forEach(c);
    }
}
```

Output

```
java -cp /tmp/ZByGrRCPkT ForEachDemo
Square of 20 is 400
Square of 21 is 441
Square of 22 is 484
Square of 23 is 529
Square of 19 is 361
Square of 17 is 289
Square of 24 is 576
Square of 26 is 676
```

6.2.6 toArray() Method

We can use toArray() method to copy elements present in the stream into specified array.

```
Integer[] ir = l1.stream().toArray(Integer[] :: new);
for(Integer i: ir) {
        sop(i);
}
```

6.2.7 Stream.of() Method

We can also apply a stream for group of values and for arrays.

```
Stream s=Stream.of(99,999,9999,99999);
s.forEach(System.out:: println);

Double[] d={10.0,10.1,10.2,10.3};
Stream s1=Stream.of(d);
s1.forEach(System.out :: println);
```

7.. Date and Time API

Java 8 provides a new Date and Time API which provides various classes and Interfaces which can be helpful in dealing with Date and Time effectively and with best performance than existing date and time system of Java.

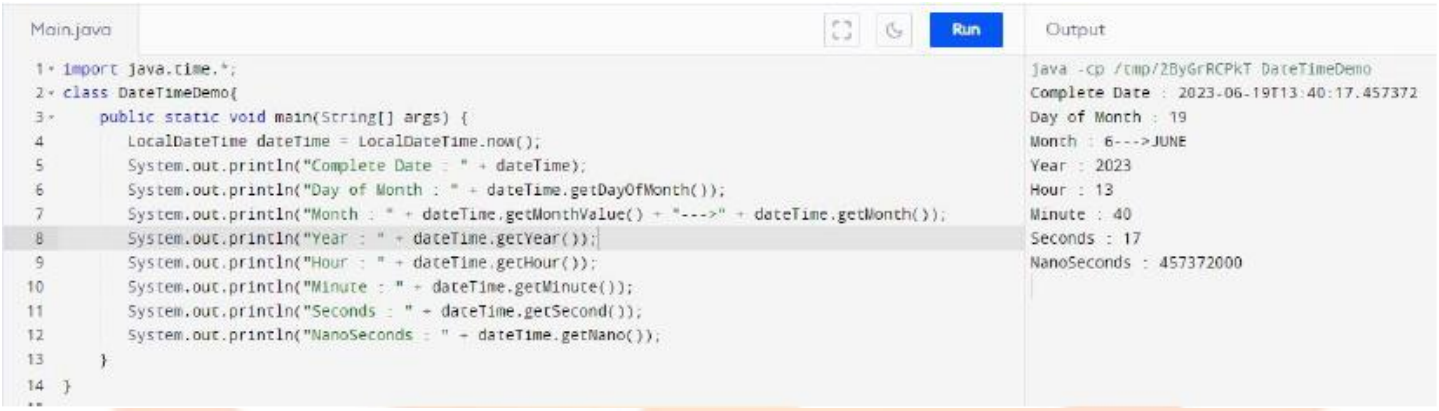
Drawbacks with Existing Java Date and Time Classes:

- The existing classes such as Date and Calendar does not provide thread safety. Hence it leads to hard-to-debug concurrency issues that are needed to be taken care by developers. The new Date and Time APIs of Java 8 provide thread safety and are immutable, hence avoiding the concurrency issue from developers.
- The classic Date and Calendar APIs does not provide methods to perform basic day-to-day functionalities. The Date and Time classes introduced in Java 8 are ISO-centric and provides number of different methods for performing operations regarding date, time, duration and periods.
- To handle the time-zone using classic Date and Calendar classes is difficult because the developers were supposed to write the logic for it. With the new APIs, the time-zone handling can be easily done with Local and ZonedDate/Time APIs.

Various Classes of Date and Time API of Java 8 are:

- LocalDate
- LocalTime
- LocalDateTime
- MonthDay
- OffsetTime
- OffsetDateTime
- Clock
- ZonedDateTime
- ZoneId
- ZoneOffset
- Year
- YearMonth
- Period
- Duration
- Instant
- DayOfWeek Enum
- Month Enum

Instead of LocalDate and LocalTime class separately, we can happily go for LocalDateTime.



We can represent a particular Date as well by using LocalDateTime class.

We can check for before and after dates as well by using this api.

Period Class

Returns the number of days, months and years between two dates.

Year Class

It provides various methods to perform on Year like checking Leap Year.

Main.java

Run

Output

```
import java.time.*;

class YearDemo {
    public static void main(String[] args) {
        Year y1 = Year.of(2000);
        Year y2 = Year.of(2021);
        System.out.println(y1.isLeap());
        System.out.println(y2.isLeap());
    }
}
```

```
java -cp /tmp/skOfVecEmh YearDemo
true
false
```

ZoneId Class

It returns the Zone in which system is available.

Main.java

Run

Output

```
import java.time.*;

class ZoneDemo {
    public static void main(String[] args) {
        ZoneId z1 = ZoneId.systemDefault();
        ZonedDateTime dt1 = ZonedDateTime.now(z1);
        System.out.println(z1);
        System.out.println(dt1);

        ZoneId z2 = ZoneId.of("Asia/Tokyo");
        ZonedDateTime dt2 = ZonedDateTime.now(z2);
        System.out.println(z2);
        System.out.println(dt2);
    }
}
```

```
java -cp /tmp/skOfVecEmh ZoneDemo
GMT
2023-06-19T17:01:58.076478Z[GMT]
Asia/Tokyo
2023-06-20T02:01:58.117613+09:00[Asia/Tokyo]
```

8.. Optional Class

The **Optional class** in Java 8 is a container object which is used to contain a value that might or might not be present. It was introduced as a way to help reduce the number of **NullPointerException** that occur in Java code. It is a part of the java.util package and was added to Java as part of Java 8.

Optional is a container that either contains a non-null value or nothing (empty Optional). The Optional class is an implementation of the Null Object pattern, which is a design pattern that was designed to reduce the number of null checks in code. With Optional, you can write cleaner and more concise code that is easier to maintain.

One of the key benefits of using Optional is that it forces you to handle the case where the value is absent. This means that you are less likely to miss important checks in your code and reduces the risk of **NullPointerException**. If a value is not present, you can either provide a default value or throw an exception.

```java
public class Main {
    public static void main(String[] args) {

        Student Student = getStudentWithName("hamza");
        System.out.println(Student.getName());

    }
    public  static Student  getStudentWithName(String name ){
        if (name.equals("hamza") || name.equals("ahmed")) {
            return new Student(name , 22 , "Morocco");
        } else {
            return null ;
        }
    }
}
```

It’s obvious that getStudentWithName() can return a null if there is no student with the given name => NullPointerException will occur when we call the getName() method (student = null).

To handle this case, we should implement some form of conditional logic based on the result returned by the method.

```java
public class Main {
    public static void main(String[] args) {
        Student Student = getStudentWithName("hamza");
        if(student != null){
            System.out.println(Student.getName());
        }else {
          System.out.println("no Student with the given name ");
        }

    }
    public  static Student  getStudentWithName(String name ){
        // lets suppose that our database contain only 2 students ahmed and hamza .
        if (name.equals("hamza") || name.equals("ahmed")) {
            return new Student(name , 22 , "Morocco");
        } else {
            return null ;
        }
    }
}
```

To avoid NullPointerException problems, let's examine what the Optional API offers.

8.1 Optional class methods

- **Optional.ofNullable()**
  To use the Optional class, simply wrap your object in an Optional object like this: Optional<T> optional = Optional.ofNullable(T). You can then use the various methods provided by Optional to handle the cases where the value is present or not. For example, the **isPresent()** method returns true if the value is present, and false if it is not.

- **Optional.of()**
  This method behave like the **ofNullable()** function it will return an Optional for the given value the only difference is that the of() method does not allow to construct an Optional object around a null value.

```
Optional<Student> student = Optional.of(null);
```

  this line will throw a NullPointerException.

- **empty()**
  this method will simply generate an empty Optional (no value).

```
Optional<T> empty = Optional.empty();
```

  The Optional class provides several methods that can be used to interact with the value that is stored inside the Optional. For example, the isPresent() method can be used to check if a value is present or not. If a value is present, you can use the get() method to retrieve it.

  Now let's try to refactor our example using Optional API.

```
1  public class Main {
2      public static void main(String[] args) {
3          Optional<Student> student = Optional.ofNullable(getStudentWithName("hamza"));
4          if (student.isPresent()) {
5              System.out.println(student.get().getName());
6          } else {
7              System.out.println("Student  is not present");
8          }
9      }
10
11     public  static Student  getStudentWithName(String name ){
12         // lets suppose that our database contain only 2 students ahmed and hamza .
13         if (name.equals("hamza") || name.equals("ahmed")) {
14             return new Student(name , 22 , "Morocco");
15         } else {
16             return null ;
17         }
18     }
19 }
```

Of course, there are still if-else blocks in this logic, but let's examine other useful methods provided by the Optional API to reduce its use. Of course, there are still if-else blocks in this logic, but let's examine other useful methods provided by the Optional API to reduce its use.

- **orElse()**
  orElse is a method in the Java Optional class that is used to return the value wrapped in the Optional, if it is present, or a default value if it is not. It takes a default value as an argument and returns it if the Optional is empty.

```
1  public class Main {
2      public static void main(String[] args) {
3          Student  student = Optional.ofNullable(getStudentWithName("hamza")).orElse(new Student("no one"
               , 0, "Unknown"));
4              System.out.println(student.getName());
5      }
6
7      public  static Student  getStudentWithName(String name ){
8          // lets suppose that our database contain only 2 students ahmed and hamza .
9          if (name.equals("hamza") || name.equals("ahmed")) {
10             return new Student(name , 22 , "Morocco");
11         } else {
12             return null ;
13         }
14     }
15 }
```

- **orElseThrow()**
  orElseThrow is a method in the Java Optional class that is used to throw an exception if the Optional is empty. Unlike orElse, which returns a default value, orElseThrow allows you to throw a custom exception. Here's an example:

```
1  public class Main {
2      public static void main(String[] args) throws StudentNotFoundException {
3          Student  student = Optional.ofNullable(getStudentWithName("fs")).orElseThrow(()-> new
               StudentNotFoundException("the Student is not Present "));
4              System.out.println(student.getName());
5      }
6
7      public  static Student  getStudentWithName(String name ){
8          // lets suppose that our database contain only 2 students ahmed and hamza .
9          if (name.equals("hamza") || name.equals("ahmed")) {
10             return new Student(name , 22 , "Morocco");
11         } else {
12             return null ;
13         }
14     }
15 }
```

public static <T> Optional<T> empty()	It returns an empty Optional object. No value is present for this Optional.
public static <T> Optional<T> of(T value)	It returns an Optional with the specified present non-null value.
public static <T> Optional<T> ofNullable(T value)	It returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional.
public T get()	If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.
public boolean isPresent()	It returns true if there is a value present, otherwise false.
public void ifPresent(Consumer<? super T> consumer)	If a value is present, invoke the specified consumer with the value, otherwise do nothing.
public Optional<T> filter(Predicate<? super T> predicate)	If a value is present, and the value matches the given predicate, return an Optional describing the value, otherwise return an empty Optional.
public <U> Optional<U> map(Function<? super T,? extends U> mapper)	If a value is present, apply the provided mapping function to it, and if the result is non-null, return an Optional describing the result. Otherwise return an empty Optional.
public <U> Optional<U> flatMap(Function<? super T,Optional<U> mapper)	If a value is present, apply the provided Optional-bearing mapping function to it, return that result, otherwise return an empty Optional.
public T orElse(T other)	It returns the value if present, otherwise returns other.
public T orElseGet(Supplier<? extends T> other)	It returns the value if present, otherwise invoke other and return the result of that invocation.
public <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier) throws X extends Throwable	It returns the contained value, if present, otherwise throw an exception to be created by the provided supplier.
public boolean equals(Object obj)	Indicates whether some other object is "equal to" this Optional or not. The other object is considered equal if: o It is also an Optional and; o Both instances have no value present or; o the present values are "equal to" each other via equals().
public int hashCode()	It returns the hash code value of the present value, if any, or returns 0 (zero) if no value is present.
public String toString()	It returns a non-empty string representation of this Optional suitable for debugging. The exact presentation format is unspecified and may vary between implementations and versions.