Java 8 Version Features:
—————————————

Up to JAVA7 version, JAVA is an Object oriented Programming Language, by understanding the advantages functional programming languages features JAVA people want to introduce functional programming features in JAVA, JAVA 8 version has the number of functional programming features.

Note: The main Advantage of Functional Programming is Code Reduction[Less Code].

1. Static Methods in Interfaces
2. Default Methods in Interfaces
3. Functional Interfaces
4. Lambda Expressions
5. Method References
6. Constructor References
7. Stream API
8. Date Time API

—————
—————

Static Methods in Interfaces:
—————————————

Up to JAVA7 version, interfaces are able to allow only abstract methods, but in JAVA 8 version we are able to write static methods inside the interfaces , if we declare static methods inside the interfaces then we must provide implementation for that methods inside the same interface only , If we want to access static methods of the interfaces in main class then we have to use only interface name, not possible to use interface reference variable, implementation class name, implementation class reference variable.

EX:

```
interface I{
void m1();
void m2();
static void m3(){
System.out.println("m3-I");
}
static void m4(){
System.out.println("m4-I");
}
```

```
}
class A implements I{
public void m1(){
System.out.println("m1-A");
}
public void m2(){
System.out.println("m2-A");
}
}
public class Main {
public static void main(String[] args) {
I i = new A();
i.m1();
i.m2();

A a = new A();
a.m1();
a.m2();

//i.m3(); ---> Error
//a.m3();----> Error
//A.m3();----> Error
I.m3();
I.m4();
}
}
```

```
m1-A
m2-A
m1-A
m2-A
m3-I
m4-I
```

EX:

```
interface I{
static void m1(){
```

```
System.out.println("m1-I");
}
static void m2(){
System.out.println("m2-I");
}
static void m3(){
System.out.println("m3-I");
}
static void m4(){
System.out.println("m4-I");
}
}

public class Main {
public static void main(String[] args) {
I.m1();
I.m2();
I.m3();
I.m4();
}
}
```

```
m1-I
m2-I
m3-I
m4-I
```

Default Methods in Interfaces

—----------------------------

Up to JAVA 7 version, interfaces are able to allow only abstract methods, but in JAVA 8 version , interfaces are able to allow default methods along with abstract methods.

In interfaces, if we declare default methods then we must provide implementation for these methods inside the interfaces only thAt we can override in the implementation classes as per the requirement.

In interfaces, we are able to declare default methods by using the “default” keyword.

IN general, we will use interfaces to declare the services that are abstract methods and these services must be implemented by the implementation classes, where if we want to provide default implementation for the services then we have to use default methods inside the interfaces that implementation classes can override or reuse.

EX:

```
interface Driver{
default void getDriverClass(){
System.out.println("sun.jdbc.odbc.JdbcOdbcDriver");
}
default void getDriverURL(){
System.out.println("jdbc:odbc:nag");
}
}
class OracleDriver implements Driver{
public void getDriverClass(){
System.out.println("oracle.jdbc.OracleDriver");
}
public void getDriverURL(){
System.out.println("jdbc:oracle:thin:@localhost:1521:xe");
}
}
class MySQLDriver implements Driver{
public void getDriverClass(){
System.out.println("com.mysql.cj.jdbc.Driver");
}
public void getDriverURL(){
System.out.println("jdbc:mysql://localhost:3300/durgadb");
}
}
class MSAccessDriver implements Driver{

}
public class Main {
public static void main(String[] args) {
Driver oracleDriver = new OracleDriver();
```

```
oracleDriver.getDriverClass();
oracleDriver.getDriverURL();
System.out.println();

Driver mysqlDriver = new MySQLDriver();
mysqlDriver.getDriverClass();
mysqlDriver.getDriverURL();
System.out.println();

Driver msaccessDriver = new MSAccessDriver();
msaccessDriver.getDriverClass();
msaccessDriver.getDriverURL();
}
}
```

```
oracle.jdbc.OracleDriver
jdbc:oracle:thin:@localhost:1521:xe

com.mysql.cj.jdbc.Driver
jdbc:mysql://localhost:3300/durgadb

sun.jdbc.odbc.JdbcOdbcDriver
jdbc:odbc:nag
```

In general, in interfaces, we are able to extend more than one interface to a single interface, if we have default methods in the super interfaces then we can access members of those super interfaces in the sub interface.

EX:

```
interface I1{
default void m1(){
System.out.println("m1-I1");
}
}
interface I2{
default void m2(){
System.out.println("m2-I2");
}
}
interface I3 extends I1, I2{
```

```
default void m3(){
System.out.println("m3-I3");
m1();
m2();
}
}
class A implements I3{

}
public class Main {
public static void main(String[] args) {
I3 i3 = new A();
i3.m3();
}
}
```

```
m3-I3
m1-I1
m2-I2
```

In the above example, if we provide the same default method with different implementations in both the super interfaces and if we access that method in the sub interface and if we get output from one default method then we can conclude that the multiple inheritance is possible in java, but it is not possible even in this situation, because if we extend more than one interface to a single interface which has the same default method then the compiler will raise an error.

EX:

```
interface I1{
default void m1(){
System.out.println("m1-I1");
}
}
interface I2{
default void m1(){
System.out.println("m1-I2");
}
}
interface I3 extends I1, I2{
default void m3(){
System.out.println("m3-I3");
```

```
m1();

}
}
class A implements I3{

}
public class Main {
public static void main(String[] args) {
I3 i3 = new A();
i3.m3();
}
}
```

Status: Compilation Error

Functional Interfaces:
—-----------------------
In Java , if we declare an interface without any abstract method then that interface is called a marker interface, in Java applications, if we declare any interface with exactly one abstract method, not more than one and not less than one then that interface is called Functional Interface.

EX:
1. Runnable: run() method
2. ActionListener: actionPerformed()
3. Comparable: compareTo()
—-----
—------

If we declare any interface with exactly one method then it is by default functional interface, but if we want to declare functional interfaces explicitly then we must use @FunctionalIterface annotation just above of the interface declaration.

EX:
```
@FunctionalInterface
interface I{
void m1();
//void m2(); ---> Error
}
class A implements I{
```

```
public void m1(){
System.out.println("m1-A");
}
public void m2(){
System.out.println("m2-A");
}
}
public class Main {
public static void main(String[] args) {
I i = new A();
i.m1();
}
}
```

In general, in Java applications, we are able to use functional interfaces to access lambda expressions.

EX:

```
@FunctionalInterface
interface I1{
void m1();
}
interface I2 extends I1{

}
```

Here I1, I2 are functional interfaces.

EX:

```
@FunctionalInterface
interface I1{
void m1();
}
interface I2 extends I1{
void m2();
}
```

Here only I1 is functional interface, I2 is not functional Interface

EX:

```
@FunctionalInterface
interface I1{
```

```
void m1();
}
@FunctionalInterface
interface I2 extends I1{

}
```

Status: No Compilation Error

EX:

```
@FunctionalInterface
interface I1{
void m1();
}
@FunctionalInterface
interface I2 extends I1{
void m2();
}
```

Status: Compilation Error

EX:

```
@FunctionalInterface
interface I1{
void m1();

}
@FunctionalInterface
interface I2 extends I1{
void m1();
}
```

Status: Valid

In Java applications, it is possible to provide a number of default methods and static methods inside the Functional Interfaces along with a single abstract method.

```
@FunctionalInterface
interface I1{
void m1();
default void m2(){
System.out.println("m2-I");
```

```
}
static void m3(){
System.out.println("m3-I");
}
}
```

Status: Valid

Lambda Expressions:
—--------------------
Lambda Expression is an anonymous Function, it does not have name, return type , access modifiers, where the method declaration and method body must be separated with -> symbol.

Syntax:
FunctionalInterface refVar = (Params) -> { —- };

To access Lambda Expressions we have to use the Functional interface Reference variable.

refVar.methodName(paramValues);

EX:
Consider the Following Program before Lambda Expressions:

```
interface Wish{
public String sayHello(String name);
}
class WishImpl implements Wish{
public String sayHello(String name){
String message = "Hello "+name+", Good Morninig!";
return message;
}
}
public class Main {
public static void main(String[] args) {
Wish wish = new WishImpl();
System.out.println(wish.sayHello("Durga"));
}
}
```

The above program with Lambda Expression:

```
interface Wish{
public String sayHello(String name);
}

public class Main {
public static void main(String[] args) {
Wish wish = (String name) -> {
String message = "Hello "+name+", Good Morninig!";
return message;
};
System.out.println(wish.sayHello("Durga"));
}
}
```

IN Lambda Expressions, parameter Data types are not mandatory.

```
interface Wish{
public String sayHello(String name);
}

public class Main {
public static void main(String[] args) {
Wish wish = (name) -> {
String message = "Hello "+name+", Good Morninig!";
return message;
};
System.out.println(wish.sayHello("Durga"));
}
}
```

In Lambda Expressions, if we have only one parameter then () are optional. If we have no Parameters and more than one parameter then it is mandatory to use ().

```
interface Wish{
public String sayHello(String name);
```

```
}

public class Main {
public static void main(String[] args) {
Wish wish = name -> {
String message = "Hello "+name+", Good Morninig!";
return message;
};
System.out.println(wish.sayHello("Durga"));
}
}
```

In Lambda Expressions body, if we have single statement then it is optional to use {} braces, in the case of single statement if we want to use return statement then it is mandatory to use {}, in this case, we can remove both {} and return statement from Lambda Expression body , if we provide any value as Lambda expression body that provided value will be returned automatically.

```
interface Wish{
public String sayHello(String name);
}

public class Main {
public static void main(String[] args) {
Wish wish = name -> "Hello "+name+", Good Morninig!";
System.out.println(wish.sayHello("Durga"));
}
}
```

Note: In Java, Lambda Expressions are applicable for only Functional Interfaces.

EX:
Before Lambda Expressions:

```
interface Welcome{
public void sayWelcome();
}
class WelcomeImpl implements Welcome{
```

```
public void sayWelcome(){
System.out.println("Welcome To Durgasoft!");
}
}
public class Main {
public static void main(String[] args) {
Welcome welcome = new WelcomeImpl();
welcome.sayWelcome();
}
}
```

After Lambda Expressions:

```
interface Welcome{
public void sayWelcome();
}
public class Main {
public static void main(String[] args) {
Welcome welcome = () -> System.out.println("Welcome To
Durgasoft!");
welcome.sayWelcome();
}
}
```

EX:

```
interface Math{
String evenOrOdd(int num);
}
public class Main {
public static void main(String[] args) {
Math math = num -> num%2==0? num+" Is Even Number": num+" is
ODD Number";
System.out.println(math.evenOrOdd(10));
System.out.println(math.evenOrOdd(5));
}
}
```

EX:

```
interface Math{
int biggest(int num1, int num2);
}
public class Main {
public static void main(String[] args) {
Math math = (num1, num2) -> num1-num2>=0?num1:num2;
System.out.println("Biggest : "+math.biggest(10,20));
System.out.println("Biggest : "+math.biggest(20,10));
}
}
```

Note:IN Java applications, we are able to pass Lambda Expressions as parameters to the methods.

Lambda Expressions in GUI Applications:

—----------

Without Lambda Expressions:

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
class MyActionListenerImpl implements ActionListener{
Frame frame;
public MyActionListenerImpl(Frame frame){
this.frame = frame;
}
@Override
public void actionPerformed(ActionEvent e) {
String label = e.getActionCommand();
if(label.equalsIgnoreCase("RED")){
frame.setBackground(Color.red);
}
if(label.equalsIgnoreCase("GREEN")){
frame.setBackground(Color.green);
}
```

```
if(label.equalsIgnoreCase("BLUE")){
frame.setBackground(Color.blue);
}
}
}
class ColorsFrame extends Frame {
Button b1;
Button b2;
Button b3;
public ColorsFrame(){
this.setVisible(true);
this.setSize(1000, 1000);
this.setTitle("Colors Frame");
this.setLayout(new FlowLayout());
this.addWindowListener(new WindowAdapter() {
@Override
public void windowClosing(WindowEvent e) {
System.exit(0);
}
});

MyActionListenerImpl myActionListener = new
MyActionListenerImpl(this);
Font font = new Font("arial", Font.BOLD, 30);
b1 = new Button("RED");
b1.setBackground(Color.red);
b1.setFont(font);
b1.addActionListener(myActionListener);

b2 = new Button("GREEN");
b2.setBackground(Color.green);
b2.setFont(font);
b2.addActionListener(myActionListener);

b3 = new Button("BLUE");
```

```
b3.setBackground(Color.blue);
b3.setFont(font);
b3.addActionListener(myActionListener);

this.add(b1);
this.add(b2);
this.add(b3);



}

}
public class Main {
public static void main(String[] args) {
ColorsFrame colorsFrame = new ColorsFrame();
}
}
```

With lambda Expressions:

```
import java.awt.*;

import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

class ColorsFrame extends Frame {
Button b1;
Button b2;
Button b3;
public ColorsFrame(){
this.setVisible(true);
this.setSize(1000, 1000);
this.setTitle("Colors Frame");
this.setLayout(new FlowLayout());
this.addWindowListener(new WindowAdapter() {
@Override
public void windowClosing(WindowEvent e) {
```

```
System.exit(0);
}
});


Font font = new Font("arial", Font.BOLD, 30);
b1 = new Button("RED");
b1.setBackground(Color.red);
b1.setFont(font);
b1.addActionListener(e -> this.setBackground(Color.red));

b2 = new Button("GREEN");
b2.setBackground(Color.green);
b2.setFont(font);
b2.addActionListener(e -> this.setBackground(Color.green));

b3 = new Button("BLUE");
b3.setBackground(Color.blue);
b3.setFont(font);
b3.addActionListener(e -> this.setBackground(Color.blue));

this.add(b1);
this.add(b2);
this.add(b3);


}

}
public class Main {
public static void main(String[] args) {
ColorsFrame colorsFrame = new ColorsFrame();
}
}
```

Lambda Expressions in Multi Threading:

Without Lambda Expressions:

```
class WelcomeThread implements Runnable{
@Override
public void run() {
for (int i = 0; i < 10; i++){
System.out.println("Welcome To Durgasoft");
}
}
}
public class Main {
public static void main(String[] args) {
WelcomeThread welcomeThread = new WelcomeThread();
Thread thread = new Thread(welcomeThread);
thread.start();


}
}
```

With Lambda Expressions:

```

public class Main {
public static void main(String[] args) {
new Thread(() -> {
for (int i = 0; i < 10; i++){
System.out.println("Welcome To Durgasoft");
}
}).start();


}
}
```

Lambda Expressions in Collections:

Without Lambda expressions:

```
import java.util.Comparator;
import java.util.TreeSet;
class MyComparator implements Comparator<String>{


@Override
public int compare(String str1, String str2) {

return str1.length() - str2.length();
}
}
public class Main {
public static void main(String[] args) {
MyComparator myComparator = new MyComparator();
TreeSet<String> treeSet = new TreeSet<String>(myComparator);
treeSet.add("AAAAA");
treeSet.add("BB");
treeSet.add("CCCC");
treeSet.add("DDD");
treeSet.add("E");
System.out.println(treeSet);


}
}
```

With Lambda Expressions

```
import java.util.Comparator;
import java.util.TreeSet;

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

```
TreeSet<String> treeSet = new TreeSet<String>((str1,str2) -> -
(str1.length() - str2.length()));
treeSet.add("AAAAA");
treeSet.add("BB");
treeSet.add("CCCC");
treeSet.add("DDD");
treeSet.add("E");
System.out.println(treeSet);


}
}
```

Method References:
—----------------
In general, in Java applications, if we declare a function interface then we are able to provide implementation for the functional interface method either by using implementation class or by using anonymous inner class or by using Lambda Expressions, where Lambda Expressions are suggestible.

In Java applications, if we want to reference a method of a class from a Functional interface method then we have to use Method Reference.

Syntax:

a) If the method is non static method
   FunctionalInterface refVar = ClassRefVar::methodName;

b) If the method is a Static method:
   FunctionalInterface refVar = ClassName::methodName;

After the Method References, if we access the Functional method by using the function interface reference variable then the mapped methods are executed internally.

```
interface Welcome{
void wish();
}
class Hello{
public void sayHello(){
```

```
System.out.println("Hello User, Welcome To Durgasoft....");
}
}
class Hi{
public static void sayHi(){
System.out.println("Hi User, Welcome to Durgasoft.....");
}
}
public class Main {
public static void main(String[] args) {
Hello hello = new Hello();
Welcome welcome1 = hello :: sayHello;
welcome1.wish();

Welcome welcome2 = Hi :: sayHi;
welcome2.wish();
}
}
```

Constructor Reference:
—----------------------
IN general, in Java applications, for the functional interface we are able to provide implementation for the abstract method by providing an implementation class or by using anonymous inner class or by using a lambda expression.

For the Functional interface method , if we want to reference a particular class constructor from the Functional interface method then we have to use Constructor reference.

Syntax:
FunctionalInterface refVar = ClassName::new;

If we access the Functional interface method by using the Functional interface reference variable then the mapped constructor will be executed.

```
interface Welcome{
void wish();
}
class Hello{
```

```
public Hello(){
System.out.println("Hello class Constructor.....");
}
}
public class Main {
public static void main(String[] args) {
Welcome welcome = Hello :: new;
welcome.wish();
}
}
```

Note: In the Functional interface method if we have any parameter then the referenced methods or constructors must have the parameter of the same type.

EX:

```
interface Welcome{
void wish(String name);
}
class Hello{
public Hello(){

}
public Hello(String name){
System.out.println("Hello class Constructor with the Parameter
Value "+name);
}
public void wishHello(String name){
System.out.println("Hello "+name+", Welcome to Durgasoft.....");
}
public static void wishHi(String name){
System.out.println("Hi "+name+", Welcome to Durgasoft......");
}
}
public class Main {
public static void main(String[] args) {
Welcome welcome1 = Hello :: new;
welcome1.wish("Durga");

Welcome welcome2 = new Hello() :: wishHello;
welcome2.wish("Nag");
```

```
Welcome welcome3 = Hello :: wishHi;
welcome3.wish("Ramesh");
}
}
```

```
Hello class Constructor with the Parameter Value Durga
Hello Nag, Welcome to Durgasoft.....
Hi Ramesh, Welcome to Durgasoft......
```

Predefined Functional Interfaces in JAVA 8 Version:

—----------

Java 8 Version has provided the following predefined functional interfaces in order to use in java applications.

1. Predicate
2. Function
3. Consumer
4. Supplier

On the basis of the above Functional interfaces some other Functional interfaces are defined in Java 8 version.

5. BiPredicate
6. BiFunction
7. BiConsumer
8. DoublePredicate
9. DoubleFunction
10. DoubleConsumer

—-----

—-----

Predicate:

—----------

Consider the below program

```
interface Math{
public boolean test(int num);
}
public class Main {
public static void main(String[] args) {
Math math = num -> num%2==0;
System.out.println("10 is Even Number? : "+math.test(10));
System.out.println("5 is Even Number? : "+math.test(5));
```

```
}
}
```

```
10 is Even Number?    : true
5 is Even Number?     : false
```

In the above program, we have declared Math as a functional interface with test() method having one parameter and the boolean return type, here the purpose of the test method is to check a particular conditional expression and return the result of the conditional expression.

IN Java applications, to evaluate a conditional expression and to return the result of the conditional expression no need to declare any Functional interface explicitly, JAVA 8 version has provided a predefined functional interface in the form of java.util.function.Predicate with the test() method like below.

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

```
import java.util.function.Predicate;
/*
interface Predicate<T>{
public boolean test(T t);
}
*/
public class Main {
public static void main(String[] args) {
Predicate<Integer> predicate = num -> num%2==0;
System.out.println("10 is Even Number? : "+predicate.test(10));
System.out.println("5 is Even Number? : "+predicate.test(5));
}
}
```

In the case of Predicates , we are able to perform the join operations by using the methods like and(), or(), negate(),...

EX:

```
import java.util.function.Predicate;

public class Main {
```

```
public static void main(String[] args) {
Predicate<Integer> predicate1 = num -> num <= 50;
Predicate<Integer> predicate2 = num -> num%2==0;

Predicate<Integer> predicate3 =
predicate1.and(predicate2);
System.out.println(predicate3.test(20));
System.out.println(predicate3.test(15));
System.out.println();

Predicate<Integer> predicate4 =
predicate1.or(predicate2);
System.out.println(predicate4.test(20));
System.out.println(predicate4.test(15));
System.out.println(predicate4.test(75));
System.out.println();

Predicate<Integer> predicate5 = predicate1.negate();
System.out.println(predicate5.test(25));
System.out.println(predicate5.test(70));

}
}
```

Function:
—----------
Consider the following program

```
interface Math{
   public int sqrRoot(int num);
}
public class Main {
   public static void main(String[] args) {
       Math math = num -> (int) java.lang.Math.sqrt(num);
       System.out.println("sqrt of 16  : "+math.sqrRoot(16));
```

```
        System.out.println("sqrt of 9  : "+math.sqrRoot(9));
    }
}
```

In the above program, we have declared Math interface with a method sqrRoot() to perform sqrRoot operation over the provided input parameter and it is able to return the resultant sqrRoot value.

To perform a particular operation over the provided parameter value and to return the resultant value of the operation no need to declare any Functional interface with an abstract method explicitly, to achieve this requirement JAVA 8 version has provided a predefined Functional interface in the form of java.util.function.Function like below.

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

EX:

```
import java.util.function.Function;

/*
public interface Function<T,R>{
public R apply(T t);
}
*/
public class Main {
public static void main(String[] args) {
Function<Double, Double> function = num ->
Math.sqrt(num);
System.out.println("sqrt of 16 :
"+function.apply(16.0d));
System.out.println("sqrt of 9 : "+function.apply(9.0D));
}
}
```

EX:

```
import java.util.function.Function;
```

```
/*
public interface Function<T,R>{
public R apply(T t);
}
*/
public class Main {
public static void main(String[] args) {
Function<String, String> function = str ->
str.concat("@durgasoft.com");
System.out.println(function.apply("abc"));
System.out.println(function.apply("xyz"));
}
}
```

Consumer:
Consumer is a functional interface, it will have a method “accept()”, it is able to take an argument and it is able to process that argument and it will not return any value.

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

EX:

```
import java.util.function.Consumer;

public class Main {
   public static void main(String[] args) {
       Consumer<String> consumer = str ->
System.out.println(str.toUpperCase());
       consumer.accept("durga software solutions");

   }
}
```

Supplier:
Supplier is a function interface in Java 8 version, it has a method “get()” , it will take no argument but it will return a value.

```
public interface Supplier<R>{
      public R get();
}
```

Note: IN general, the Supplier will be utilized in Java applications in order to supply values to the applications.
EX: To generate OTPs or Dynamic Port Numbers, Secrete codes,.... We will use Supplier.

EX:

```
import java.util.function.Consumer;
import java.util.function.Supplier;

public class Main {
   public static void main(String[] args) {
        Supplier<Integer> supplier = () ->
(int)(Math.random()*1000000);
        System.out.println(supplier.get());
        System.out.println(supplier.get());
        System.out.println(supplier.get());
        System.out.println(supplier.get());

   }
}
```

BiPredicate is the same as Predicate, it has a test() method containing two arguments, it will perform a condition with two arguments and it will return a result.

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

EX:

```
import java.util.function.BiPredicate;

public class Main {
```

```
   public static void main(String[] args) {
        BiPredicate<Integer, Integer> biPredicate = (num1,num2) -
> num1 > num2;
        System.out.println("10 > 20 ? : "+biPredicate.test(10,20)
);
        System.out.println("20 > 10 ? : "+biPredicate.test(20,10)
);

   }
}
```

Similarly, we can analyze BiFunction, BiConsumer, here BiFunction and BiConsumer are the same as the Function and Consumer but these functional interfaces methods will take two arguments.

EX:

```
import java.util.function.BiConsumer;
import java.util.function.BiFunction;

public class Main {
   public static void main(String[] args) {
        BiFunction<Integer, Integer, Float> biFunction = (num1,
num2) -> (float) (num1+num2)/2;
        System.out.println(biFunction.apply(12,23));

        BiConsumer<String, String> biConsumer = (str1,str2) ->
System.out.println(str1+str2);
        biConsumer.accept("Durga", "soft");



   }
}
```

To deal with Double values in the Functional INterfaces, JAVA 8 version has provided the functional interfaces like

DoublePredicate
DoubleFunction
DoubleConsumer

These Functional interfaces are the same as the Predicate, Function and Consumer but these functional interfaces methods will take double as an argument type.

EX:

```
import java.util.function.DoubleConsumer;
import java.util.function.DoubleFunction;
import java.util.function.DoublePredicate;

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

       DoublePredicate doublePredicate = num -> num < 100.0d;
       System.out.println("50.0d < 100.0d   ? :
"+doublePredicate.test(50.0d));
       System.out.println("200.0d < 100.0d  ? :
"+doublePredicate.test(200.0d));

       DoubleFunction<Double> doubleFunction = num ->
Math.sqrt(num);
       System.out.println(doubleFunction.apply(16.0d));
       System.out.println(doubleFunction.apply(64.0d));

       DoubleConsumer doubleConsumer = num ->
System.out.println(num*num);
       doubleConsumer.accept(10.0);
       doubleConsumer.accept(20.0);
   }
}
```

Streams:
—-------
Stream is an Object, it is able to collect the objects from any Collection and it is able to process those objects as per the application requirements.

EX: To separate only even numbers from the numbers which are available in a Collection object.

EX: To generate email Ids from the names which are available in a Collection object.

Q)What is the difference between Streams from java.io package and Stream from java.util.stream ?

—----------

Ans:

—---

Stream in java.io package is channel or medium, it is able to canary the data from input devices to java applications and from java applications to the output devices, here Streams are able to carry the data in the form of characters or binary.

Stream in java.util.stream package is an object , it is able to collect the objects from any Collection and it is able to process those objects as per the application requirements, here Stream is able to process objects.

Q)What is the difference between Collection and Stream?

—----------

Ans:

—--

Collection is an object, it is able to represent a group of other objects as a single entity.

Stream is an object, it is able to get a group of objects from a Collection object in order to process the objects.

In Java applications, if we want to use Streams then we have to use the following steps.

1. Create a Stream object from Collection.
   public Stream stream()

   EX: Stream stream = arrayList.stream();

2. Configure Stream either with filter or with map:
   After creating Stream , we have to configure the Stream in the following two ways.
   1. filter
   2. map

   If we want to configure a Stream with filter then we have to use the following method from Stream.
   public Stream filter(Predicate p)
   EX: Stream<Integer> s = al.stream().filter(num->num<10);

If we want to configure a Stream with Map then we have to use the Following method from Stream.

public Stream map(Function f)
EX: Stream<String> s = al.stream().map(name->name+"@dss.com");

3. Process the Stream:
   To Process the Stream elements we have to use the following Methods.
   1. collect()
   2. count()
   3. sorted()
   4. min()
   5. max()
   6. forEach()
   7. toArray()
   8. of()

collect():
This method is able to collect all the elements from Stream.
EX: Filter
Without Streams:

```
import java.util.ArrayList;
import java.util.List;

public class Main {
   public static void main(String[] args) {
       List<Integer> list =  new ArrayList<Integer>();
       for(int i = 1; i <= 100; i++){
           list.add(i);
       }
       List<Integer> evenList = new ArrayList<Integer>();
       for(int num: list){
           if(num % 2 == 0){
               evenList.add(num);
           }
       }
       System.out.println(evenList);
```

```
    }
}
```

With Streams:

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Main {
    public static void main(String[] args) {
        List<Integer> list =  new ArrayList<Integer>();
        for(int i = 1; i <= 100; i++){
            list.add(i);
        }
        /*
        List<Integer> evenList = new ArrayList<Integer>();
        for(int num: list){
            if(num % 2 == 0){
                evenList.add(num);
            }
        }
        System.out.println(evenList);
         */
        List<Integer> evenList = list.stream().filter(num-
>num%2==0).collect(Collectors.toList());
        System.out.println(evenList);



    }
}
```

EX: map:

Without Streams:

```java
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("durga");
        list.add("ramesh");
        list.add("balayya");
        list.add("santosh");
        list.add("Prakash");
        list.add("suresh");
        list.add("pavankalyan");

        List<String> emailList = new ArrayList<String>();
        for (String name: list){
            name = name + "@durgasoft.com";
            emailList.add(name);
        }

        for (String email: emailList){
            System.out.println(email);
        }


    }
}
```

With Streams:

```java
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("durga");
```

```
        list.add("ramesh");
        list.add("balayya");
        list.add("santosh");
        list.add("Prakash");
        list.add("suresh");
        list.add("pavankalyan");
        /*
        List<String> emailList = new ArrayList<String>();
        for (String name: list){
            name = name + "@durgasoft.com";
            emailList.add(name);
        }

         */
        List<String> emailList = list.stream().map(name-
>name+"@durgasoft.com").collect(Collectors.toList());

        for (String email: emailList){
            System.out.println(email);
        }


    }
}
```

count():

—---------

It is able to count the number of elements in the Stream object.

EX:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;

public class Main {
   public static void main(String[] args) {
        List<String> list = new ArrayList<>();
```

```
        list.add("durga");
        list.add("ramesh");
        list.add("balayya");
        list.add("santosh");
        list.add("Prakash");
        list.add("suresh");
        list.add("pavankalyan");

      long l = list.stream().filter(name-
>!name.startsWith("s")).count();
        System.out.println(l);

    }
}
```

sorted():
It is able to sort all the elements in the Stream object in a default sorting order.

EX:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;

public class Main {
   public static void main(String[] args) {
        List<String> list = Arrays.asList("Balayya",
"Chiru", "Prabhas", "Mahesh", "Pavan Kalyan");
      List<String> sortedList =
list.stream().sorted().collect(Collectors.toList());
        System.out.println(sortedList);

    }
}
```

If we want to provide customized sorting order over the Stream elements then we have to pass Comparator as a parameter to the sorted() method.

EX:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;

public class Main {
   public static void main(String[] args) {
       List<String> list = Arrays.asList("Balayya",
"Chiru", "Prabhas", "Mahesh", "Pavan Kalyan");
       List<String> sortedList =
list.stream().sorted((str1,str2)->-
str1.compareTo(str2)).collect(Collectors.toList());
       System.out.println(sortedList);

   }
}
```

min() and max():

---

min() is able to return minimum value over the sorted elements in the Stream.
max() is able to return maximum value over the sorted elements in the Stream.

EX:

```
import java.util.Arrays;
import java.util.List;

public class Main {
   public static void main(String[] args) {
       List<Integer> list = Arrays.asList(5, 2, 7, 1, 8,
4, 9, 3, 10, 6);
       int minVal = list.stream().min((num1, num2)->
num1-num2).get();
       System.out.println(minVal);
```

```
        int maxVal = list.stream().max((num1,num2)->(num1-
num2)).get();
        System.out.println(maxVal);

    }
}
```

forEach:
It is able to apply an operation over each and every element of the Stream.

EX:

```
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(5, 2, 7, 1, 8,
4, 9, 3, 10, 6);
        //list.stream().forEach(num->
System.out.println(num));
        list.stream().forEach(System.out::println);

    }
}
```

toArray():
It is able to convert all the elements to an Array.
EX:

```
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(5, 2, 7, 1, 8,
4, 9, 3, 10, 6);
      Object[] objArray = list.stream().toArray();
```

```
        for (Object val: objArray){
            System.out.println(val);
        }
    }
}
```

of():
It is able to create a Stream object on the basis of the elements.
EX:

```
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Main {
    public static void main(String[] args) {
        Stream<Integer> stream =
Stream.of(10,20,30,40,50);

System.out.println(stream.collect(Collectors.toList()));
    }
}
```

Q)What is the difference between map() and flatMap()?
—--
Ans:
—---
In Stream API, map(Function) method is able to take a Function as parameter and it is able to transfer elements from one Stream to Another Stream, if the element includes any Collection then it is able to transfer that collection as it is to the target Stream.

In Stream API, flatMap(Function) method is able to take a Function as parameter and it will transfer the elements from one Stream to another Stream , if the element contains any Collection then it will convert the collection to Stream, here flatMap() will get all individual elements from Stream and it will send that individual elements to the target Stream, here the process of Converting streams of elements to a single stream is called “Flattering”.

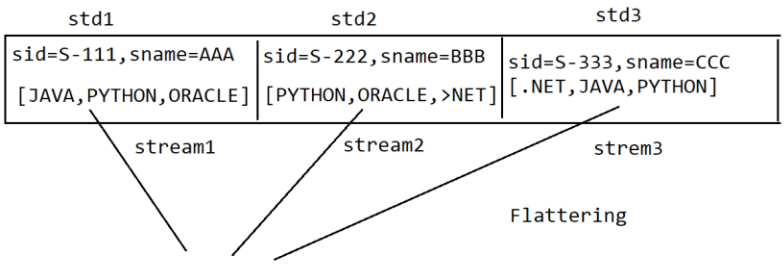
EX:

```
import java.util.Arrays;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

class Student{
   private String sid;
   private String sname;
   private String saddr;
   private List<String> coursesList;

   public Student(String sid, String sname, String saddr,
List<String> coursesList) {
       this.sid = sid;
       this.sname = sname;
       this.saddr = saddr;
       this.coursesList = coursesList;
   }

   public String getSid() {
       return sid;
   }

   public void setSid(String sid) {
       this.sid = sid;
   }

   public String getSname() {
       return sname;
   }

   public void setSname(String sname) {
       this.sname = sname;
   }

   public String getSaddr() {
       return saddr;
   }

   public void setSaddr(String saddr) {
```

```
        this.saddr = saddr;
    }

    public List<String> getCoursesList() {
        return coursesList;
    }

    public void setCoursesList(List<String> coursesList) {
        this.coursesList = coursesList;
    }
}
public class Main {
    public static void main(String[] args) {

        Student std1 = new Student("S-111", "Durga", "Hyd",
Arrays.asList("JAVA", "PYTHON", "AWS"));
        Student std2 = new Student("S-222", "Anil", "Pune",
Arrays.asList("JAVA", "ORACLE", ".NET"));
        Student std3 = new Student("S-333", "Nag", "Delhi",
Arrays.asList("PYTHON", "MYSQL", "DEVOPS"));
        Student std4 = new Student("S-444", "Ramesh", "Chennai",
Arrays.asList("JAVA", "DJANGO", "LINUX"));

        List<Student> stdList = Arrays.asList(std1, std2,std3,
std4);

        List<String> sidList = stdList.stream().map(std-
>std.getSid()).collect(Collectors.toList());
        System.out.println(sidList);

        List<String> snameList = stdList.stream().map(std-
>std.getSname()).collect(Collectors.toList());
        System.out.println(snameList);

        List<String> saddrList = stdList.stream().map(std-
>std.getSaddr()).collect(Collectors.toList());
        System.out.println(saddrList);

        List<List<String>> stdCoursesList =
stdList.stream().map(std-
>std.getCoursesList()).collect(Collectors.toList());
```

```
        System.out.println(stdCoursesList);

        List<String> coursesList = stdList.stream().flatMap(std-
>std.getCoursesList().stream()).collect(Collectors.toList());
        System.out.println(coursesList);

        Set<String> coursesSet = stdList.stream().flatMap(std-
>std.getCoursesList().stream()).collect(Collectors.toSet());
        System.out.println(coursesSet);



    }
}
```

```
[S-111, S-222, S-333, S-444]
[Durga, Anil, Nag, Ramesh]
[Hyd, Pune, Delhi, Chennai]
[[JAVA, PYTHON, AWS], [JAVA, ORACLE, .NET], [PYTHON, MYSQL, DEVOPS], [JAVA,
DJANGO, LINUX]]
[JAVA, PYTHON, AWS, JAVA, ORACLE, .NET, PYTHON, MYSQL, DEVOPS, JAVA, DJANGO,
LINUX]
[JAVA, DJANGO, DEVOPS, MYSQL, LINUX, .NET, PYTHON, AWS, ORACLE]
```



flatMap() [JAVA, PYTHON, ORACLE, PYTHON, ORACLE, .NET, .NET, JAVA,PYTHON] stream

map()[[JAVA,PYTHON,ORACLE],[PYTHON,ORACLE,.NET],[.NET,JAVA,PYTHON]]

Q)What is the difference between map() and reduce()?

—--
Ans:
—---
map() is able to process each and every element with an operation and return the result of that element to another stream.

reduce() is able to perform Aggregate operations like sum, sub, mul, avg, min, max,.... Over the stream elements and generate the results directly in the form of Optional object, if we access get() method over the Optional object then we are able to get the required value.

```
import java.util.Arrays;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

public class Main {
   public static void main(String[] args) {
       List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9);
       System.out.println(list);
       List<Integer> list1 = list.stream().map(num-
>num*2).collect(Collectors.toList());
       System.out.println(list1);

       int sum = list.stream().reduce((a, b)->a+b).get();
       System.out.println(sum);

       int mul = list.stream().reduce((a, b)->a*b).get();
       System.out.println(mul);

       int min = list.stream().reduce(Integer::min).get();
       System.out.println(min);

       int max = list.stream().reduce(Integer::max).get();
       System.out.println(max);
   }
}
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 4, 6, 8, 10, 12, 14, 16, 18]
45
```

362880
1
9

Date Time API:
—--------------
The Date Time API was provided by the JAVA 8 version in order to represent Date values more effectively than the java.util.Date class.

Java 8 version has provided the complete predefined library for the Date and time representation in the form of java.time package.

To represent Date and Time in java applications we have to use the following predefined classes.

1. LocalDate
2. LocalTime
3. LocalDateTime
   —----

To create LocalDate , LocalTime, LocalDateTime objects we will use the following method from all the above classes.

public static XXX now()

If we want to get Date details from LocalDate class we have to use the following methods.

public DayOfWeek getDayOfWeek()
public int getDayOfMonth()
public int getMonthValue()
public int getYear()

EX:

```
import java.time.DayOfWeek;
import java.time.LocalDate;

public class Main {
```

```
    public static void main(String[] args) {
        LocalDate localDate = LocalDate.now();
        System.out.println(localDate);

        DayOfWeek day = localDate.getDayOfWeek();
        System.out.println(day);

        int date = localDate.getDayOfMonth();
        System.out.println(date);

        int month = localDate.getMonthValue();
        System.out.println(month);

        int year = localDate.getYear();
        System.out.println(year);

        System.out.println(day+"
"+date+"/"+month+"/"+year);


    }
}
```

If we want to get time details like hour, minute, second, nano second then we have to use the following methods from LocalTime class.

```
public int getHour()
public int getMinite()
public int getSecond()
public long getNano()
```

EX:

```
import java.time.LocalTime;

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

```
        LocalTime localTime = LocalTime.now();
        System.out.println(localTime);
        System.out.println("Hour        :
"+localTime.getHour());
        System.out.println("Minute      :
"+localTime.getMinute());
        System.out.println("Second      :
"+localTime.getSecond());
        System.out.println("Nano Sec    :
"+localTime.getNano());

    }
}
```

If we want to get both Date and Time values at a time we have to use LocalDateTime class.

EX:

```
import java.time.LocalDateTime;
import java.time.LocalTime;

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

        LocalDateTime localDateTime = LocalDateTime.now();
        System.out.println(localDateTime);
        System.out.println("Day         :
"+localDateTime.getDayOfWeek());
        System.out.println("Date        :
"+localDateTime.getDayOfMonth());
        System.out.println("Month       :
"+localDateTime.getMonthValue());
        System.out.println("Year        :
"+localDateTime.getYear());
        System.out.println("Hour        :
"+localDateTime.getHour());
```

```
        System.out.println("Minute      :
"+localDateTime.getMinute());
        System.out.println("Second      :
"+localDateTime.getSecond());
        System.out.println("Nano Sec    :
"+localDateTime.getNano());

    }
}
```

In all the above classes, we are able to represent our own Date, time and Date time by using a static method of().

EX:

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;

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

        LocalDate localDate = LocalDate.of(2020, 12, 10);
        System.out.println(localDate);

        LocalTime localTime = LocalTime.of(10, 25, 56,
123456);
        System.out.println(localTime);

        LocalDateTime localDateTime =
LocalDateTime.of(localDate, localTime);
        System.out.println(localDateTime);

    }
}
```

java.time.Period:
It is able to represent a time duration.

To create a Period object we have to use the following static method from Period class.

public static Period between(LocalDate date1, LocalDate date2)

To get the number of years, months and days we have to use the following methods from Period class.

public int getYears()
public int getMonths()
public int getDays()

EX:

```
import java.time.LocalDate;
import java.time.Period;

public class Main {
   public static void main(String[] args) {
        LocalDate localDateNow = LocalDate.now();
        LocalDate localDateDOB = LocalDate.of(1999,11,25);
        Period period = Period.between(localDateDOB,
localDateNow);
        System.out.println("AGE    : "+period.getYears()+" Years
"+period.getMonths()+" Months and "+period.getDays()+" Days");

   }
}
```

AGE   : 23 Years 8 Months and 7 Days

ZoneId:
—-------
ZoneId is able to represent a particular Time zone.

If we want to get the current System time then we have to use the following method from ZoneId .

public static systemDefault()

If we want to represent a particular time zone we have to use the following static method from ZoneId.

public static ZoneId of(String timezone_Code)

If we want to get Date and time on the basis of a particular time zone we have to use the following method from the ZonedDateTime class.

Public static ZonedDateTime now(ZoneId zoneId)

EX:

```
import java.time.ZoneId;
import java.time.ZonedDateTime;

public class Main {
   public static void main(String[] args) {
       ZoneId zoneId = ZoneId.systemDefault();
       System.out.println(zoneId);

       ZoneId newZoneId = ZoneId.of("America/Los_Angeles");
       ZonedDateTime zonedDateTime =
ZonedDateTime.now(newZoneId);
       System.out.println(zonedDateTime);
   }
}
```