Java 8 Features:
---
1. Static Methods in interfaces
2. Default Methods in interfaces
3. Functional Interfaces
4. Lambda Expressions
5. Predicates and Functions
6. Method Reference
7. Constructor Reference
8. Stream API
9. Date Time API

1. Static Methods in interfaces

The main intention of introducing static methods in interfaces is to improve sharability.

Upto JAVA7 version, interfaces are able to allow only abstract methods, but, from JAVA8 version interfaces are able to allow static methods.

In interfaces, if we declare static methods then we must provide implementation part for that method at the same declaration.

If we declare static methods inside an interface then we are able to access that static method by using the respective interface name only, not possible to access with interface reference variable, implementation class name and implementation class reference variable.

EX:

```
package java8features;

interface I{
	static void m1() {
		System.out.println("m1-A");
	}
	static void m2() {
		System.out.println("m2-A");
	}
}
class A implements I{

}
public class Test {

	public static void main(String[] args) {
		I.m1();
		I.m2();
		//A.m1(); ---> Error
		//A.m2(); ---> Error

		I i = new A();
		//i.m1(); ----> Error
		//i.m2(); ----> Error
```

```
                A a = new A();
                //a.m1(); ---> Error
                //a.m2(); ---> Error
        }
}
```

2. Default Methods in interfaces

Upto JAVA7 version, interfaces are able to include only abstract methods, but, JAVA8 version onwards, interfaces are able to allow default methods , which includes initial / default implementation.

In general, in java applications, we will use interfaces to declare services and we will give an option to implement these services to some other module members or some other thirds party vendors, in this context, to provide initial / default implementation for the service methods inside the interfaces then we have to use "default methods".

If we declare default methods inside the interfaces then we are able to reuse that default implementation or we may override the default implementation as per our application requirement.

To declare default methods inside the interfaces we have to use "default" keyword.

Note: If we want to add any new functionality to all the implementation classes with out manipulating implementation classes then we have to use "default" methods inside the interface.
EX:

```
package java8features;

interface DB_Driver{// SUN Microsystems
        default void getDriverClass() {
                System.out.println("sun.jdbc.odbc.JdbcOdbcDriver");
        }
        default void getDriverURL() {
                System.out.println("jdbc:odbc:nag");
        }
}
class OracleDriver implements DB_Driver{// Oracle
        public void getDriverClass() {
                System.out.println("oracle.jdbc.OracleDriver");
        }
        public void getDriverURL() {

System.out.println("jdbc:oracle:thin:@localhost:1521:xe");
        }
}

class MySQLDriver implements DB_Driver{// MySQL
        public void getDriverClass() {
                System.out.println("com.mysql.jdbc.Driver");
        }
        public void getDriverURL() {

System.out.println("jdbc:mysql://localhost:3306/durgadb");
```

```
	}
}

class MSAccessDriver implements DB_Driver{// MS Access

}
public class Test {

	public static void main(String[] args) {
		DB_Driver oracleDriver = new OracleDriver();
		oracleDriver.getDriverClass();
		oracleDriver.getDriverURL();
		System.out.println();
		DB_Driver mysqlDriver = new MySQLDriver();
		mysqlDriver.getDriverClass();
		mysqlDriver.getDriverURL();
		System.out.println();
		DB_Driver msaccessDriver = new MSAccessDriver();
		msaccessDriver.getDriverClass();
		msaccessDriver.getDriverURL();
	}
}
```

3. Functional Interfaces:

If we have any interface with exactly one abstract method then that interface is called as Functional Interface.
EX: java.lang.Runnable
    java.util.Comparable
    java.awt.event.ActionListener

If we want to declare user defined interface as functional interface then we have to use the following annotation.

```
@FunctionalInterface
public interface Persistable{
----
}
```
Note: In the case of Functional Interfaces, we are able to write any no of static methods and any no of default methods in side the interfaces, but, only one abstract method.
EX:
```
----
package java8features;

@FunctionalInterface
interface I{
	void m1();
	//void m2(); --> Error
	static void m2() {
		System.out.println("m2-I");
	}
	static void m3() {
		System.out.println("m3-I");
	}
	default void m4() {
		System.out.println("m4-I");
```

```
        }
        default void m5() {
                System.out.println("m5-I");
        }
}
class A implements I{
        public void m1() {
                System.out.println("m1-A");
        }
}
public class Test {

        public static void main(String[] args) {
                I i = new A();
                i.m1();
                I.m2();
                I.m3();
                i.m4();
                i.m5();
        }
}
```

4. Lambda Expressions

--> Lambda Expressions is a feature derived from Lambda Calculas, It was introduced in 1930's in Maths, by getting the advantages of Lambda Calc. Programming Languages are implementing Lambda Expressions.
EX: C, VC++, c#.net, C++, Python, Ruby, Java,...

--> Lambda Expressins are providing the following advantages in Java applications.
1.It will reduce Code.
2.It can be used as an alternative for Anonymous Inner classes.
3.It can be passed as parameters to the methods.
4.It will introduce Function /Procedure oriented programming in Java
5.It will be used as an Object.
6.It provides replacement for the implementation classes of the Functinal interfaces.

--> Lambda Expression is an anonymous Function, it is a normal function, it does not include name, return type and access modifiers.
EX:

```
public void add(int i, int j)
{
   System.out.println(i+j);
}
```

Equalent Lambda Expression:

```
(int i, int j) -> {
                        System.out.println(i+j);
                }
```

EX:

```
public void sayHello(String name){
   System.out.println("Hello "+name+"!");
```

```
}
Equalent Lambda Expression:
(String name) -> {
                   System.out.println("Hello "+name+"!");
                  }
```

--> If we want to access Lambda Expressions then we have to use Functional interface.
EX:

```
package java8features;
interface Calculator{
        public void add(int i, int j);
}
public class Test {
public static void main(String[] args) {
Calculator cal = (int i, int j) -> {
                                  System.out.println(i+j);
                                  };
cal.add(10, 20);
}
}
```

--> In Lambda Expression body, if we are using only one statement then curly braces are optional.
EX:

```
interface Calculator{
        public void add(int i, int j);
}
public class Test {

        public static void main(String[] args) {
                Calculator cal = (int i, int j) ->
System.out.println(i+j);
                cal.add(10, 20);
        }
}
```

--> In Lambda Expression , If the paramter types predicted by the compiler automatically depending on the situation then parameter data types are optional.
EX:

```
interface Calculator{
        public void add(int i, int j);
}
public class Test {

        public static void main(String[] args) {
                Calculator cal = (i,j) -> System.out.println(i+j);
                cal.add(10, 20);
        }
}
```

--> In Lambda Expression, if only one parameter is existed then it is optional to provide paramthesys[()]

EX:

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

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

--> In Lambda Expressions, it is possible to return values like normal java methods.

EX:

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

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

Note: If we want to use "return" keyword in Lambda Expression Body then curly braces are mandatory, if we remove return keyword then curly braces are not required and provide the value at right side of the expression with out using curly braces then JVM will return the provided value from the Lambda Expression.

EX:

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

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

Lambda Expressions as Replacement for Anonymous Inner Classes:

EX with out Anonymous Inner class:

```
class MyThread implements Runnable{
	public void run() {
		for(int i = 0; i < 10; i++) {
			System.out.println("User Thread :"+i);
```

```
                }
        }
}
public class Test {
        public static void main(String[] args) {
                MyThread mt = new MyThread();
                Thread t = new Thread(mt);
                t.start();
        }
}
```

EX for Anonymous Inner class:

```
----------------------------
public class Test {
        public static void main(String[] args) {
                Runnable r = new Runnable() {
                        public void run() {
                                for(int i = 0; i < 10; i++) {
                                        System.out.println("User
thread :"+i);
                                }
                        }
                };
                Thread t = new Thread(r);
                t.start();
        }
}
```

EX for Lambda Expressions:

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

                new Thread(() -> {
                        for(int i = 0; i < 10; i++) {
                                System.out.println("User Thread :"+i);
                        }
                }).start();

        }
}
```

Lambda Expressions in GUI Programming

```
-------------------------------------
package com.durgasoft.core;

import java.awt.Button;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.Frame;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

class ColorsFrame extends Frame {
```

```
        Button b1, b2, b3;

        public ColorsFrame() {
                this.setVisible(true);
                this.setSize(500, 500);
                this.setTitle("Colors Frame");
                this.setLayout(new FlowLayout());
                this.addWindowListener(new WindowAdapter() {
                        @Override
                        public void windowClosing(WindowEvent e) {
                                System.exit(0);
                        }
                });

                b1 = new Button("RED");
                b2 = new Button("GREEN");
                b3 = new Button("BLUE");

                Font f = new Font("consolas", Font.BOLD, 15);
                b1.setFont(f);
                b2.setFont(f);
                b3.setFont(f);


                b1.addActionListener(ae ->
this.setBackground(Color.red));
                b2.addActionListener(ae ->
this.setBackground(Color.green));
                b3.addActionListener(ae ->
this.setBackground(Color.blue));



                this.add(b1); this.add(b2); this.add(b3);
        }
}
public class Test {
        public static void main(String[] args) {
                ColorsFrame frame = new ColorsFrame();

        }
}
```

Lambda Expressions in Collections

```
package com.durgasoft.core;

import java.util.TreeSet;

public class Test {
        public static void main(String[] args) {
                StringBuffer sb1 = new StringBuffer("AAA");
                StringBuffer sb2 = new StringBuffer("BBBBB");
                StringBuffer sb3 = new StringBuffer("C");
                StringBuffer sb4 = new StringBuffer("DDDD");
                StringBuffer sb5 = new StringBuffer("EE");
```

```
                TreeSet<StringBuffer> ts = new TreeSet<StringBuffer>
((s1, s2) -> ((StringBuffer)s1).length() -
((StringBuffer)s2).length());
                ts.add(sb1);
                ts.add(sb2);
                ts.add(sb3);
                ts.add(sb4);
                ts.add(sb5);
                System.out.println(ts);
        }
}
```

5. Predicates and Functions

Predicate:

--> Predicate is a predefined functional interface provided by JAVA8 version as java.util.function.Predicate.

--> Predicate functional interface is having test() method, it will take an input parameter and it will check the provided condition and it will return the result of the conditional expression , that is, true or false value.

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

--> In Java applications, if we want to test a condition and if we want to return the result of test case then prepare Lambda Expression for Predicate interface and use that Lambda expression in Java application, it will reduce our explicit condition checking and return statements.

Example : Write a Predicate to check whether the given number is less than 10 or not.

With out Lambda Expression:

```
public interface Predicate{
  public boolean test(int i);
}
class MyPredicate implements Predicate{
public boolean test(int i){
  if(i < 10 ){
    return true;
  }else{
    return false;
  }
}
}

class Test{
public static void main(String[] args){
Predicate p = new MyPredicate();
System.out.println(p.test(15));// false
System.out.println(p.test(5));// true
```

```
}
}
```

With Lambda Expression:

```
class Test{
public static void main(String[] args){
Predicate p = i -> (i<10);

System.out.println(p.test(15));// false
System.out.println(p.test(5));// true
}
}
```

Function:

--> Function is a a Functional interface provided by JAVA 8 Version as java.util.function.Function.

--> Function is a functional interface, it includes apply(--) method, it will take an input parameter of any data type and it will return a value of any data type[not only boolean].

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

--> IN Java8 version, Predicate is mainly for Condition checking and Function is mainly for processing over the data.

Example: Calculate length of the String which we provided as an input.

With out Lambda Expression:

```
public interface Function{
public int apply(String str);
}
class MyFunction implements Function{
public int apply(String str){
  return str.length();
}
}
class Test{
public static void main(String[] args){
Function f = new MyFunction();
SYstem.out.println(f.apply("abc"));// 3
System.out.println(f.apply("Durgasoft"));// 9
}
}
```

With Lambda Expression:

```
class Test{
public static void main(String[] args){
Function f = str -> str.length();
SYstem.out.println(f.apply("abc"));// 3
System.out.println(f.apply("Durgasoft"));// 9
}
}
```

--> In Java applications, we are able to join more than one Predicate by using the functions like add(), or(), negate()
EX:

```
package java8features;

import java.util.function.Predicate;

public class Test {
	public static void main(String[] args) {
		Predicate<Integer> p1 = i -> (i < 10);
		Predicate<Integer> p2 = i -> (i%2==0);

		Predicate<Integer> p3 = p1.negate();
		System.out.println(p3.test(6));// false

		Predicate<Integer> p4 = p1.and(p2);
		System.out.println(p4.test(6));// true

		Predicate<Integer> p5 = p1.or(p2);
		System.out.println(p5.test(5));// true

	}
}
```

:: operator / Method References:

The main intention of :: operator or Method References is to map a normal java method from the Function interface method, in this context, if we access Functional interface method then JVM will execute the respective mapped Java method.
Syntaxes:

If the mapped method is an instance method :
```
   Fun_Interface ref_Var = Class_Ref_Var :: Method_Name;
```

If the mapped method is a static method :
```
  Fun_Interface ref_Var = Class_Name :: Method_Name;
```

In the above situations, if we access Functional interface method by using Functional Interface reference variable then JVM will execute the mapped method.
Example:

```
package java8features;

@FunctionalInterface
interface I{
	void m1();
}
class A{
	void m2() {
		System.out.println("m2-A");
	}
	static void m3() {
```

```
                System.out.println("m3-A");
        }
}
public class Test {
        public static void main(String[] args) {
                A a = new A();
                I i = a :: m2;
                i.m1();

                I i1 = A :: m3;
                i1.m1();

        }
}
```

:: Operator / Constructor References:

The main intention of Constructor references is to map Functional interface method with a particular class constructor, in this context, if we access Functional interface method then JVM will execute the mapped constructor.
Note: If we have parameters in the functional interface method then same type of parameters must be existed in thye mapped constructors.

Syntax:

```
Fun_Interface ref_Var = Class_Name :: new;
```

Example:

```
package java8features;

@FunctionalInterface
interface I1{
        void m1();
}

@FunctionalInterface
interface I2{
        void m2(int i);
}

@FunctionalInterface
interface I3{
        void m3(float f);
}
class A{
        A(){
                System.out.println("A-0-arg-con");
        }
        A(int i){
                System.out.println("A-int-param-con :"+i);
        }
        A(float f){
                System.out.println("A-float-param-con :"+f);
        }
```

```
}
public class Test {
        public static void main(String[] args) {
                I1 i1 = A :: new;
                i1.m1();

                I2 i2 = A :: new;
                i2.m2(10);

                I3 i3 = A :: new;
                i3.m3(22.22f);
        }
}
```

6. Stream API

IN Java applications, Collection objects are able to represent a group of Other objects as a single entity, but, Stream API is a set of predefined Classes and Interfaces, which are used to process the elements which are represented in the form of Collection objects.

Note: In Java, IO Streams are able to process binary data and character data, but, Stream API is able to process objects.

The complete predefined classes and interfraces which are required for Stream API are provided by JAVA8 version in the form of java.util.stream package.

If we want to use Stream API in Java applications then we have to use the following steps.

1. Create Stream object from a particular Collection
2. Configure Stream object.
3. Processing the elements.

1. Create Stream object from a particular Collection

```
    public Stream stream();
    EX: Stream s = al.stream();
```

2. Configure Stream object:

There are two ways to configure Stream object.

1. By using Filter
2. By Using Map

Where Filtering mechanism will use a Predicate to filter the elements, for this we have to use the following method from java.util.stream.Stream.

```
   public Stream filter(Predicate p)
```

Where Map mechanism will use a Function to process the elements, for this we have to use the following method from java.util.stream.Stream

```
   public Stream map(Function f)
```

3. Processing the elements.

After getting all the elements in the form of Stream we have to proccess that elements by using the following methods.

1. collect(-)
2. count(-)
3. sorted(-)
4. min(-)
5. max(-)
6. forEach(-)

Example on the basis of Filter:

```
package java8features;


import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Test {
        public static void main(String[] args)throws Exception {
                ArrayList<Integer> list = new ArrayList<Integer>();
                for(int i = 1; i <= 10; i++) {
                        list.add(i);
                }
                // Processing for Even numbers
                Stream<Integer> s1 = list.stream();

                Predicate<Integer> p = i -> (i%2 == 0);
                Stream<Integer> s2 = s1.filter(p);

                List<Integer> l = s2.collect(Collectors.toList());
                System.out.println(l);

                // Processing for odd numbers
                Stream<Integer> s3 = list.stream();
                Stream<Integer> s4 = s3.filter(i -> (i%2 != 0));
                List<Integer> li = s4.collect(Collectors.toList());
                System.out.println(li);

        }
}
```

Example on Map:

```
package java8features;


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

```
public class Test {
	public static void main(String[] args)throws Exception {
		ArrayList<String> al = new ArrayList<String>();
		al.add("aaa");
		al.add("BBB");
		al.add("ccc");
		al.add("DDD");
		al.add("eee");
		al.add("FFF");
		System.out.println(al);

		Stream<String> s1 = al.stream();
		Function<String, String> f = str -> str.toLowerCase();
		Stream<String> s2 = s1.map(f);
		List<String> list1 = s2.collect(Collectors.toList());
		System.out.println(list1);

		Stream<String> s3 = al.stream();
		Function<String, String> fn = str ->
str.toUpperCase();
		Stream<String> s4 = s3.map(fn);
		List<String> list2 = s4.collect(Collectors.toList());
		System.out.println(list2);

	}
}
```

2. count(-):
---> It will count the no of elements which are existed in Stream.
EX:

```
import java.util.ArrayList;
import java.util.function.Predicate;
import java.util.stream.Stream;

public class Test {
	public static void main(String[] args)throws Exception {
		ArrayList<Integer> al = new ArrayList<Integer>();
		for(int i = 1; i <= 10; i++) {
			al.add(i);
		}
		System.out.println(al);

		Stream<Integer> s1 = al.stream();
		Predicate<Integer> p = i -> (i%2 == 0);
		Stream<Integer> s2 = s1.filter(p);
		System.out.println(s2.count());

	}
}
```

3. sorted(-)

It able to keep all the elements in a particular Sorting order.
Note: By using sorted() method we are able to provide both natural

sorting or customized sorting.

Example

```
package java8features;

import java.util.ArrayList;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Test {
        public static void main(String[] args)throws Exception {
                ArrayList<String> al = new ArrayList<String>();
                al.add("ccc");
                al.add("aaa");
                al.add("fff");
                al.add("bbb");
                al.add("eee");
                al.add("ddd");
                System.out.println(al);

                Stream<String> s1 = al.stream();
                Function<String, String> fn = str ->
str.toUpperCase();
                Stream<String> s2 = s1.map(fn);
                Stream<String> s3 = s2.sorted((str1,str2) -> -
str1.compareTo(str2) );
                System.out.println(s3.collect(Collectors.toList()));

        }
}
```

4. min(-)

--> It will return min value from Stream
EX:

```
package java8features;

import java.util.ArrayList;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Test {
        public static void main(String[] args)throws Exception {
                ArrayList<String> al = new ArrayList<String>();
                al.add("ccc");
                al.add("aaa");
                al.add("fff");
                al.add("bbb");
                al.add("eee");
                al.add("ddd");
                System.out.println(al);
```

```
                Stream<String> s1 = al.stream();
                Function<String, String> fn = str ->
str.toUpperCase();
                Stream<String> s2 = s1.map(fn);
                Stream<String> s3 = s2.sorted();
                System.out.println(s3.min((str1, str2)->
str1.compareTo(str2)));


        }
}
```

5. max(-)
--> It able to return max value from Stream
EX:

```
package java8features;

import java.util.ArrayList;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Test {
        public static void main(String[] args)throws Exception {
                ArrayList<String> al = new ArrayList<String>();
                al.add("ccc");
                al.add("aaa");
                al.add("fff");
                al.add("bbb");
                al.add("eee");
                al.add("ddd");
                System.out.println(al);

                Stream<String> s1 = al.stream();
                Function<String, String> fn = str ->
str.toUpperCase();
                Stream<String> s2 = s1.map(fn);
                Stream<String> s3 = s2.sorted();
                System.out.println(s3.max((str1, str2)->
str1.compareTo(str2)));


        }
}
```

6. forEach(-)

It able to get all the elements in forEach loop manner
Ex:

```
package java8features;

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

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

public class Test {
        public static void main(String[] args)throws Exception {
                ArrayList<String> al = new ArrayList<String>();
                al.add("ccc");
                al.add("aaa");
                al.add("fff");
                al.add("bbb");
                al.add("eee");
                al.add("ddd");
                System.out.println(al);

                Stream<String> s1 = al.stream();
                Function<String, String> fn = str ->
str.toUpperCase();
                Stream<String> s2 = s1.map(fn);
                s2.forEach(System.out :: println);
        }
}
```

7. Date Time API

Date Time API from JAVA8 version is providing simlified approach to represent date and time in JAVA applications.

JAVA8 version has provided the complete predefined library in the form of "java.time" package.

java.time package has provided the following predefined classes to represent date in java applications.

java.time.LocalDate
java.time.LocaTime
java.time.LocalDateTime

Example:

```
package java8features;

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

public class Test {
        public static void main(String[] args)throws Exception {
                LocalDate date = LocalDate.now();
                System.out.println(date);
                int day = date.getDayOfMonth();
                int month = date.getMonthValue();
                int year = date.getYear();
                System.out.println(day+"-"+month+"-"+year);
                System.out.println();
```

```
                LocalTime time = LocalTime.now();
                System.out.println(time);
                int hour = time.getHour();
                int mnt = time.getMinute();
                int scn = time.getSecond();
                int nscn = time.getNano();
                System.out.println(hour+":"+mnt+":"+scn+":"+nscn);
                System.out.println();

                LocalDateTime dt = LocalDateTime.now();
                System.out.println(dt);
                int day1 = dt.getDayOfMonth();
                int month1 = dt.getMonthValue();
                int year1 = dt.getYear();
                int hour1 = dt.getHour();
                int mnt1 = dt.getMinute();
                int scn1 = dt.getSecond();
                int nscn1 = dt.getNano();
                System.out.println(day1+"-"+month1+"-
"+year1+":"+hour1+":"+mnt1+":"+scn1+":"+nscn1);


        }
}
```

Java9 Features:
---
1. Private methods in Interfaces
2. Try-With-Resources Enhancement
3. Diamand Operator In Generics
4. Unmodifiable Collections
5. JSHELL Programming
6. JPMS

   -----

1. Private methods in Interfaces:

Upto JAVA7 version, interfaces are able to allow only public and abstract methods.
In JAVA8 version, interfaces are able to allow static methods and default methods.
In JAVA9 version, interfaces are able to allow private methods inorder to improve Reusability.

IN Java9 version, we may write no of default methods, in default methods, if we have any common implementations then to reduce duplicate code and to improve code reusability we have to use private methods inside the interfaces.
EX: In Transaction interface, we may declare transaction methods like deposit, withdraw, transferAmmount,... with the common implementations like Open Database Conection, Start Transactions , Close Transactin and Close the connection. This approach will increase no of instructions unneccessarily, to redice duplicate code we will use private methods with the common implementatin and we will access that methods in our actual business methods.

EX:

```
package java9features;

interface Transaction{
	private void preTransaction() {
		System.out.println("Open Database Connection");
		System.out.println("Start Transaction");

	}
	private void postTransactions() {
		System.out.println("Close Transaction");
		System.out.println("Close Database Connection");
	}
	public default void deposit() {
		preTransaction();
		System.out.println("----Logic to perform Deposit
Operation----");
		postTransactions();
	}
	public default void withdraw() {
		preTransaction();
		System.out.println("----Logic to perform Withdraw
Operation----");
		postTransactions();
	}
	public default void transferAmmount() {
		preTransaction();
		System.out.println("----Logic to perform Transfer
Ammount Operation----");
		postTransactions();
	}
}
class TransactionImpl implements Transaction{

}
public class Test {

	public static void main(String[] args) {
		Transaction tx = new TransactionImpl();
		tx.deposit();
		System.out.println();
		tx.withdraw();
		System.out.println();
		tx.transferAmmount();
	}
}
```

2. Try-With-Resources Enhancement

If we want to use resources like database Connections, Streams,.... in Java applicatins then we have to use try-catch-finally inorder to handler the exceptions which are generated by the resources.

If we want to manage resources with try-catch-finally, JAVA has given a seperate convention.

1. Declare the resources before try.
2. Create the resources inside the try.
3. Close the resources inside finally.
EX:

```
package java9features;

import java.io.BufferedReader;
import java.io.FileOutputStream;
import java.io.InputStreamReader;

public class Test {

        public static void main(String[] args) {
                BufferedReader br = null;
                FileOutputStream fos = null;
                try {
                        br = new BufferedReader(new
InputStreamReader(System.in));
                        System.out.print("Enter Data :");
                        String data = br.readLine();
                        fos = new
FileOutputStream("E:/documents/abc.txt");
                        byte[] b = data.getBytes();
                        fos.write(b);
                } catch (Exception e) {
                        e.printStackTrace();
                }finally {
                        try {
                                br.close();
                                fos.close();
                        } catch (Exception e) {
                                e.printStackTrace();
                        }
                }
        }
}
```

In the above approach, we must close the resources explicitly and we have to perform resources closig operations inside try-catch-finally inside finally, it will increase confusion to the developers.

To overcome the above problems JAVA7 version has provided try-with-resources.
In case of try-with-resources, it is not required to close the resources, where JVM will close all the resources which are declared with try when the flow of execution is coming out from try block, but, the resources classes or interfaces must extend or implement java.lang.AutoClosable markar interface.
Syntax:

```
try(Res1; Res2; .....Res-n){
   ---
}catch(Exception_Name ref_Var){
   ---
}
```

EX:

```
package java9features;

import java.io.BufferedReader;
import java.io.FileOutputStream;
import java.io.InputStreamReader;

public class Test {

	public static void main(String[] args) {

		try(
				BufferedReader br = new
BufferedReader(new InputStreamReader(System.in));
				FileOutputStream fos = new
FileOutputStream("E:/documents/abc.txt");
		) {
			System.out.print("Enter Data :");
			String data = br.readLine();
			byte[] b = data.getBytes();
			fos.write(b);
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

IN the above try-with-resources option, we must declare the resources along with try only, it is not possible to provide resources references directly.

JAVA9 version has given flexibility to pass directly resources references as parameters to try.

```
Resource1 res1 = new Resource1();
Resource2 res2 = new Resource2();
Resource3 res3 = new Resource3();
try(res1;res2;res3){
----
}catch(Exception_Name e)
---
}
```
EX:

```
package java9features;

import java.io.BufferedReader;
import java.io.FileOutputStream;
import java.io.InputStreamReader;

public class Test {

	public static void main(String[] args)throws Exception {
		BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
		FileOutputStream fos = new
```

```
FileOutputStream("E:/documents/abc.txt");
                try(br;fos) {
                        System.out.print("Enter Data :");
                        String data = br.readLine();
                        byte[] b = data.getBytes();
                        fos.write(b);
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

3. Diamand Operator In Generics

IN general, in JAVA applications, Generics are introduced to improve Typedness in JAVA applications in JDK1.5 version onwards.

Syntax:

```
Collection_Name<Type> ref = new Collection_Name<Type>();
```

EX:

```
ArrayList<String> al = new ArrayList<String>();
al.add("AAA");
al.add("BBB");
al.add("CCC");
al.add("DDD");
al.add(new Integer(10)); ---> Error
System.out.println(al);
```

OP: [AAA, BBB, CCC, DDD]

IN JDK1.7 version, Diamond operator [<>] was introduced, it will give flexibility to the developers about to specify generic type at only Left side expression , not required to specify at right side expression in Collections objects creations, it is sufficient to specify <> at right side espression.

EX:

```
ArrayList<String> al = new ArrayList<>();
al.add("AAA");
al.add("BBB");
al.add("CCC");
al.add("DDD");
al.add(new Integer(10)); ---> Error
System.out.println(al);
```

OP: [AAA, BBB, CCC, DDD]

Upto JDK1.8 version, Diamond operator is not posisble for Annonymous inner classes, but, JDK1.9 version has given flexibility to the developers to apply diamon operator for Annonymous inner classes.

EX:

```
Interface_Name<Type> ref = new Inteface_Name<>()
{
   ---implementation-----
};
```

EX:

```
import java.util.*;
class Test
{
        public static void main(String[] args)
        {
                StringBuffer sb1 = new StringBuffer("AAAA");
                StringBuffer sb2 = new StringBuffer("BB");
                StringBuffer sb3 = new StringBuffer("CCCCC");
                StringBuffer sb4 = new StringBuffer("D");
                StringBuffer sb5 = new StringBuffer("EEE");

                Comparator<StringBuffer> c = new Comparator<>()
                {
                        public int compare(StringBuffer sb1,
StringBuffer sb2)
                        {
                                int length1 = sb1.length();
                                int length2 = sb2.length();
                                int val = 0;
                                if(length1 < length2)
                                {
                                        val = -100;
                                }
                                else if(length1 > length2)
                                {
                                        val = 100;
                                }
                                else
                                {
                                        val = 0;
                                }
                                return -val;
                        }
                };
                TreeSet<StringBuffer> ts = new TreeSet<StringBuffer>
(c);
                ts.add(sb1);
                ts.add(sb2);
                ts.add(sb3);
                ts.add(sb4);
                ts.add(sb5);
                System.out.println(ts);
        }
}
```

4. Unmodifiable Collections :

UnmodifiableList

Upto JAVA8 version, to declare List object as an immutable List object then we have to access the following method from java.util.Collections class after adding the required no of elements.

```
public List<T> unmodifiableList(List<T> list)
```

Note: After declaring List object as UnmodifiableList if we add or

modify or remove any element from the List then JVM will provide an Exceptino like " java.lang.UnsupportedOperationException".
EX:

```
import java.util.*;
class Test
{
        public static void main(String[] args)
        {
                List<String> al = new ArrayList<String>();
                al.add("AAA");
                al.add("BBB");
                al.add("CCC");
                al.add("DDD");
                System.out.println(al);
                al = Collections.unmodifiableList(al);
                al.add("EEE"); --> UnsupportedOperationException

        }
}
```

The above approach to make List object as an immutable object is difficult to implement in JAVA applications, JAVA9 version has provided stright forward factory methods to make List object as an Immutable object .

```
public List<E> of(E e)
public List<E> of(E e1, E2)
public List<E> of(E e1, E e2, E e3)
```


```
public List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)
public List<E> of(E ... e)
```

Note: If we provide any null element in the List then JVM will raise java.lang.NullPointerException.
Note: After declaring List object as an immutable object if we add/remove/modify any element then then JVM will raise an Exception like "java.lang.UnsupportedOperationException".
EX:

```
import java.util.*;
class Test
{
        public static void main(String[] args)
        {
                List<String> al = List.of("AAA", "BBB", "CCC", "DDD");
                System.out.println(al);
                //al.add("EEE");----> UnsupportedOperatinException
        //List<String> list = List.of("AAA", "BBB", "CCC", null);-->NullPointerException

        }
}
```

UnmodifiableSet

Upto JAVA8 version, to declare Set object as an immutable Set object then we have to access the following method from java.util.Collections class after adding the required no of elements.

```
public Set<T> unmodifiableSet(Set<T> set)
```

Note: After declaring Set object as UnmodifiableSet if we add or modify or remove any element from the Set then JVM will provide an Exceptino like " java.lang.UnsupportedOperationException".
EX:

```
import java.util.*;
class Test
{
        public static void main(String[] args)
        {
                Set<String> s = new HashSet<String>();
                s.add("AAA");
                s.add("BBB");
                s.add("CCC");
                s.add("DDD");
                s.add(null);
                s = Collections.unmodifiableSet(s);
                System.out.println(s);
                //s.add("EEE"); --> UnSupportedOperationException
        }
}
```

The above approach to make Set object as an immutable object is difficult to implement in JAVA applications, JAVA9 version has provided stright forward factory methods to make Set object as an Immutable object .

```
public Set<E> of(E e)
public Set<E> of(E e1, E2)
public Set<E> of(E e1, E e2, E e3)
-----
----
public Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9,
E e10)
public Set<E> of(E ... e)
```

Note: If we provide any null element in the Set then JVM will raise java.lang.NullPointerException.
Note: After declaring Set object as an immutable object if we add/remove/modify any element then then JVM will raise an Exception like "java.lang.UnsupportedOperationException".
EX:

```
import java.util.*;
class Test
{
        public static void main(String[] args)
        {
                Set<String> s1 = Set.of("AAA", "BBB", "CCC", "DDD");
                System.out.println(s1);
```

```
                //s1.add("EEE"); -->UnSupportedOperationException
        //Set<String> s2 = Set.of("AAA", "BBB", "CCC", null);--
>NullPointerException

        }
}
```

UnmodifiableMap:

Upto JAVA8 version, to declare Map object as an immutable Map object then we have to access the following method from java.util.Collections class after adding the required no of elements.

```
public Map<K,V> unmodifiableMap(Map<K,V> map)
```

Note: After declaring Map object as UnmodifiableMap if we add or modify or remove any element from the Map then JVM will provide an Exceptino like " java.lang.UnsupportedOperationException".
EX:

```
import java.util.*;
class Test
{
        public static void main(String[] args)
        {
                Map<String, String> m = new HashMap<>();
                m.put("A", "AAA");
                m.put("B", "BBB");
                m.put("C", "CCC");
                m.put("D", "DDD");
                m.put("E", null);
                m = Collections.unmodifiableMap(m);
                System.out.println(m);
        }
}
```
The above approach to make Map object as an immutable object is difficult to implement in JAVA applications, JAVA9 version has provided stright forward factory methods to make Map object as an Immutable object .

```
public Map<K,V> of(K k1, V v1)
public Map<K,V> of(K k1, V v1, K k2, V v2)
public Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3)
-----
----
public Map<K,V> of(K k1, V v1, K k2, V v2,.......K k10, V v10)
```

Note: If we provide any null element in the Map then JVM will raise java.lang.NullPointerException.
Note: After declaring Map object as an immutable object if we add/remove/modify any element then then JVM will raise an Exception like "java.lang.UnsupportedOperationException".
EX:

```
import java.util.*;
class Test
{
```

```
        public static void main(String[] args)
        {
                Map<String, String> m = Map.of("A","AAA", "B","BBB",
"C","CCC", "D","DDD");
                System.out.println(m);
                //m.put("E", "EEE"); --> UnSupportedOperationException
                //Map<String, String> m1 = Map.of("A","AAA",
"B","BBB", "C","CCC", "D",null);---->NullPointerException

        }
}
```

5. JSHELL Programming

JSHELL is a command based tech, it will be used to check the code instantly.
JSHELL will use REPL[Read Evaluate Program Loop] tool, command based tool.

JSHELL is providing env for checking our code instantly, it is not an alternative to any IDE and it is not providing any Code optimization.

To open JSHELL we have to set classpath env to JAVA9 versin and use the following on command prompt.

```
D:\java430>set path=C:\Java\jdk9.0.4\bin;

D:\java430>JSHELL
|  Welcome to JShell -- Version 9.0.4
|  For an introduction type: /help intro
jshell>
```

IN JSHELL, we can evaluate any type valid Java expression.
EX:

```
jshell> System.out.println("Welcome To JSHELL");
Welcome To JSHELL

jshell> 10+20
$2 ==> 30

jshell> int i = 10;
i ==> 10

jshell> i + 20;
$4 ==> 30

jshell> 10<15
$5 ==> true

jshell>
```

We can get Hystory in JSHELL by using the following command.

```
jshell> /history

System.out.println("Welcome To JSHELL");
10+20
```

```
int i = 10;
i + 20;
10<15
/history
```

We can get all code snippets which we entered in JSHELL by using the following command.

```
jshell> /list

   1 : System.out.println("Welcome To JSHELL");
   2 : 10+20
   3 : int i = 10;
   4 : i + 20;
   5 : 10<15
```

In JSHELL, we are able to declare variables explicity , they are called as Explicit variables. If we evaluate any expression in JSHELL, where JSHELL will store the result of the provided expression in the form of implicit variables called as Scratch variables, where scratch variables are starts with $.

To get all variables which are existed i JSHELL we have to use the following command.

```
jshell> /vars
|    int $2 = 30 ----> Implicit Variable
|    int i = 10 -----> Explicit Variable
|    int $4 = 30 ----> Implicit Variable
|    boolean $5 = true ----> implicit Variables

jshell>
```

We can declare and access methods in JSHELL.

```
jshell> void sayHello(String name){   -----------> Method Declaration
   ...> System.out.println("Hello "+name+"!");
   ...> }
|  created method sayHello(String)

jshell> sayHello("Durga") -----> Metho call
Hello Durga!

jshell> int add(int i, int j){ ---> Method declaration
   ...> int k = i + j;
   ...> return k;
   ...> }
|  created method add(int,int)

jshell> add(10,20); ---> Method call
$10 ==> 30
```

IN JSHELL, we are able to list out all the methods which we declared.

```
jshell> /methods
|    void sayHello(String)
|    int add(int,int)

jshell>
```

In JSHELL, we are able to declare the classes, interfaces and abstract

classes and we are able to create objects for the classes and we are able to access the memebrs of the classes.

```
jshell> public class Employee
   ...> {
   ...>     String eid = "E-111";
   ...>     String ename = "Durga";
   ...>     float esal = 5000;
   ...>     String eaddr = "Hyd";
   ...>     public void getEmployeeDetails()
   ...>     {
   ...>         System.out.println("Employee Details");
   ...>         System.out.println("---------------------");
   ...>         System.out.println("Employee Id     :"+eid);
   ...>         System.out.println("Employee Name   :"+ename);
   ...>         System.out.println("Employee Salary :"+esal);
   ...>         System.out.println("Employee Address:"+eaddr);
   ...>     }
   ...> }
|  created class Employee

jshell> Employee emp = new Employee();
emp ==> Employee@69ea3742

jshell> emp.getEmployeeDetails();
Employee Details
---------------------
Employee Id     :E-111
Employee Name   :Durga
Employee Salary :5000.0
Employee Address:Hyd



jshell> interface AccountService{
   ...> public void create();
   ...> public void search();
   ...> public void update();
   ...> public void delete();
   ...> }
|  created interface AccountService

jshell> class AccountServiceImpl implements AccountService{
   ...> public void create(){
   ...> System.out.println("Acount Created Successfully");
   ...> }
   ...> public void search(){
   ...> System.out.println("Account Search Success");
   ...> }
   ...> public void update(){
   ...> System.out.println("Account Updated Successfully");
   ...> }
   ...> public void delete(){
   ...> System.out.println("Account Deleted Successfully");
   ...> }
   ...> }
|  created class AccountServiceImpl
```

```
jshell> AccountService accService = new AccountServiceImpl();
accService ==> AccountServiceImpl@3c0f93f1

jshell> accService.create();
Acount Created Successfully

jshell> accService.search();
Account Search Success

jshell> accService.update();
Account Updated Successfully

jshell> accService.delete();
Account Deleted Successfully

jshell>
```

It is difficult to write or modify code in JSHELL command prompt, to simplify code manipulations JSHELL has provided its own editor, to get JSHELL editor we will use the following command.

```
jshell> /edit
```

Note: In JSHELL, we are able to set our own editors also like notepad or editplus,....

```
jshell> /set editor "C://Windows//System32//notepad.exe"
|  Editor set to: C://Windows//System32//notepad.exe

jshell> /edit   --> we will get Notpad as editor


jshell> /set editor "C://Program Files (x86)//EditPlus//editplus.exe"
|  Editor set to: C://Program Files (x86)//EditPlus//editplus.exe

jshell> /edit  --> We will get Editplus as Editor
```

In JSHELL, we are able to access members of a particular package by importing the rehspective package and bny setting classpath env variable.

To set classpath env variable we will use the following command.

```
JSHELL> /env -class-path=ClassPathValue
```

EX:

```
C:\Users\Nagoor>set path=C:\Java\jdk-9.0.4\bin;

C:\Users\Nagoor>JSHELL
|  Welcome to JShell -- Version 9.0.4
|  For an introduction type: /help intro

jshell> /env -class-
path=C:\oracle18xe\product\18.0.0\dbhomeXE\jdbc\lib\ojdbc8.jar;
```

```
|  Setting new options and restoring state.

jshell> /set editor "C://editplus//editplus.exe"
|  Editor set to: C://editplus//editplus.exe

jshell> /edit
|  created class Employee

jshell> Employee emp = new Employee();
emp ==> Employee@704deff2

jshell> emp.getEmpList();
ENO     ENAME   ESAL    EADDR
-------------------------------
111     AAA     5000.0  Hyd
222     BBB     6000.0  Hyd
333     CCC     7000.0  Hyd
444     DDD     8000.0  Hyd

Employee.java
--------------
import java.sql.*;
public class Employee
{
        Connection con;
        Statement st;
        ResultSet rs;
        public Employee(){
                try{
                        Class.forName("oracle.jdbc.OracleDriver");
                        con =
DriverManager.getConnection("jdbc:oracle:thin:@192.168.1.9:1521:xe","s
ystem","durga");
                        st = con.createStatement();
                }catch(Exception e){
                        e.printStackTrace();
                }
        }
        public void getEmpList(){
                try{
                        rs = st.executeQuery("select * from emp1");
                        System.out.println("ENO\tENAME\tESAL\tEADDR");
                        System.out.println("-------------------------
-----");
                        while(rs.next()){

System.out.print(rs.getInt("ENO")+"\t");

System.out.print(rs.getString("ENAME")+"\t");

System.out.print(rs.getFloat("ESAL")+"\t");

System.out.print(rs.getString("EADDR")+"\n");
                        }
                }catch(Exception e){
                        e.printStackTrace();
                }finally{
```

```
                        try{
                                rs.close();
                                st.close();
                                con.close();
                        }catch(Exception e){
                                e.printStackTrace();
                        }
                }
        }
}
```

6. JPMS[Java Platform Module System]

Q)What is the requirement to go for Mudule System over Jars and packages?

Ans:

Drawbacks are existed with JARs:
1. Unexpected NoClassDefFoundError
2. Version Conflicts
3. Low Security
4. Bigger Size Library

1. Unexpected NoClassDefFoundError:
--> In Enterprise applications, we are able to use more and more no of JAR files as per the requirement, if we want to use all these JAR files we must set "classpath" env variable to all these JAR files, while putting all these JAR files if any JAR file is missing then JVM is able to raise an Exception like java.lang.NoClassDefFoundError" in the middle of the application execution.

2. Version Conflicts
--> IN Enterprise applications, we are able to use no of JAR files for, which may be designed in the same versin or may be in different versions, if all the JAR files existed in different versions there may be a chance to get exceptions in the middle.
Note: If we more than JAR file contains the same class which is required in our main application then JVM will provide some Exception in application execution.

3. Low Security:
--> In java application development, if we use JAR files then that JAR files content is open to all , all the developers can use the content of JAR files, it will provide less security for the content.

4. Bigger Library Size :
--> Upto JAVA8 version, the complete predefined library is existed in rt.jar, if we want to prepare any simple Java application then we must load the complete rt.jar file, which containe 4000+ no of classes and interfaces upto JAVA8 version with the 80mb, it is making all java applications as Heavy weight applications.

To overcome all the above problems, we have to go for JPMS provided by

JAVA9 version.

In JPMS, the complete library is existed in the form of modules.
java.base
java.rmi
java.sql
java.net

Note: In JAVA9 version, we are able to prepare our own modules with our own library.

Module: Module is a folder containes packages and a special file called as module-info.java , it is module configuration file.

Module = Packages + Module Configuration File.

Steps to prepare java application in JPMS:
---
0. Prepare Application Folder in our System.
1. Create src folder in our application folder.
2. Create Module folder under src folder
3. Create Java application with a package
4. Create module-info.java file
5. Compile Module
6. Execute Module

1. Create src folder in our application fiolder:

D:\java9
---
app1
|---src

2. Create Module folder under src flder
---
app1
|---src
     |---moduleA

3. Create Java application with a package

Test.java

```
package com.durgasoft.core;
public class Test
{
        public static void main(String[] args)
        {
                System.out.println("Welcome To JPMS");
        }
}
```

4. Create module-info.java file

```
---------------------------------
module-info.java
----------------
module moduleA{
}

5. Compile Module
-----------------
D:\java9\app1>set path=C:\Java\jdk-9.0.4\bin;
D:\java9\app1>javac --module-source-path src -d out -m moduleA

6. Execute Module
-----------------
D:\java9\app1>java --module-path out -m
moduleA/com.durgasoft.core.Test
Welcome To JPMS
-------------------------------------------------------------------------
-------
In JPMS, if we want to export any package to out side of the present
module we have to use "exports" attribute in module-info.java.

In JPMS, if we want to access other modules in the present Module we
have to use "requires" attribute in module-info.java file.
EX:
module-info.java
----------------
module moduleA{
exports com.durgasoft.core
requires moduleB
}

Example:
D:\java9
---------
app2
|---src
|    |----moduleA
|    |       |-----Test.java
|    |       |-----module-info.java
|    |----moduleB
             |-----Employee.java
             |-----module-info.java

moduleA/Test.java
-----------------
package com.durgasoft.test;
import com.durgasoft.emp.Employee;
public class Test
{
        public static void main(String[] args)
        {
                Employee emp = new Employee("E-111", "AAA", 5000,
"Hyd");
                emp.getEmpDetails();
        }
}
```

moduleA/module-info.java

```
module moduleA{
        requires moduleB;
}
```

moduleB/Employee.java

```
package com.durgasoft.emp;
public class Employee
{
        String eid;
        String ename;
        float esal;
        String eaddr;

        public Employee(String eid, String ename, float esal, String
eaddr)
        {
                this.eid = eid;
                this.ename = ename;
                this.esal = esal;
                this.eaddr = eaddr;
        }
        public void getEmpDetails()
        {
                System.out.println("Employee Details");
                System.out.println("-----------------------");
                System.out.println("Employee Id       :"+eid);
                System.out.println("Employee Name     :"+ename);
                System.out.println("Employee Salary   :"+esal);
                System.out.println("Employee Address  :"+eaddr);
        }
}
```

moduleB/module-info.java

```
module moduleB{
        exports com.durgasoft.emp;
}
```